Technical Report
CMU/SEI-92-TR-036
ESD-TR-92-036

Durra: A Task Description Language User's Manual (Version 2)

Dennis L. Doubleday
Mario R. Barbacci

December 1992

Technical Report
CMU/SEI-92-TR-036
ESD-TR-92-036
December 1992

Durra: A Task Description Language User's Manual (Version 2)

Dennis L. Doubleday
Mario R. Barbacci

Distributed Systems

Unlimited distribution subject to the copyright

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1992 by Carnegie Mellon University.

Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

Requests for permission to reproduce this document or to prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTIBILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

1 Introduction 5

2 A Durra Development Scenario 9
2.1 Component Creation Activities 10
2.2 Application Creation Activities 10
2.3 Application Execution Activities 11

3 Getting Started with Durra 13
3.1 Environment Variables 13
3.2 Internet Service Definition Requirements 13
3.3 Durra System Directory Structure 14
3.4 File Naming Conventions 15
3.5 Descriptive Conventions in This Document 16

4 Durra Tools and Runtime Library 17
4.1 The Library Management Utility 17
4.2 The Compilation Utility 18
4.3 The Compiler 18
4.4 The Cluster Directory Population Utility 19
4.5 The Application Recompilation Utility 20
4.6 The Runtime Library 20
4.7 The Predefined Channel Library 21
4.8 The Cluster Activation Daemon (Durra_Launcher) 21

5 Example : An Informal Durra Application Construction Process 23
5.1 The Example Application 23

6 Writing a Durra Application Description 25
6.1 Type Declarations 25
6.2 Port Specifications 25
6.3 Behavioral Specifications 25
6.4 Attribute Specifications 25
6.5 Primitive Component Descriptions 26
6.6 Compound Task Descriptions 26

7 Writing a Durra Task Implementation 31
7.1 The Durra Application Programmer's Interface 31
7.1.1 Types and Constants 31
7.1.2 Exceptions 32
7.1.3 Time Query Operations 32
7.1.4 ***Init*** and ***Finish*** Operations 32

7.1.5 Process Attribute Query Operation 33
7.1.6 Port and Type Identification Query Operations 33
7.1.7 Port Status Query Operations 34
7.1.8 Input Operations 34
7.1.9 Output Operations 35
7.1.10 Reconfiguration Support Operations 36
7.2 Example Usage of the ***Durra_Interface*** Package 37
7.2.1 The Producer Implementation 38
7.2.2 The Consumer Implementation 39

8 Writing a Durra Channel Implementation 43
8.1 The Channel Package Specification Template 43
8.2 The Channel Package Body Template 45
8.3 Example: Our ***FIFO_Channel*** Implementation 47

9 Durra Configuration Files 53

10 Compiling and Linking a Durra Application 55
10.1 Durra Compilation and Makefiles 55
10.2 Cluster Compilation 57

11 Executing a Durra Application 59

Appendix A VADS Dependencies in Durra 61

Appendix B The *Durra_Interface* Package Specification 63

References 73

Index 75

List of Figures

Figure 1-1	A Template for Task Descriptions	6
Figure 2-1	Durra Application Development Scenario	9
Figure 3-1	The Durra System Directory Structure	14
Figure 5-1	Dynamic Producer-Consumer Application Structure	24
Figure 6-1	A Primitive Durra Task Description	26
Figure 6-2	Generic Channel and Task Descriptions	27
Figure 6-3	A Producer-Consumer Application Description	27
Figure 6-4	A Producer-Consumer Application Description Featuring Dynamic Reconfiguration	28

Durra: A Task Description Language User's Manual (Version 2)

Abstract: This document describes the use of Durra, a task-level application description language, and its associated toolset. The Durra environment supports the development of highly reconfigurable distributed Ada applications. The intended audience for this document is system managers responsible for Durra installation and Durra application developers.

1 Introduction

Computing environments consisting of loosely connected networks of special and general purpose processors are becoming commonplace. The corresponding trend in software is away from sequential programs running on large uniprocessor hardware, and toward concurrent programs distributed across multiple, possibly heterogeneous, platforms. Today, developers of such applications typically "hard code" the allocation of computing resources into their applications by explicitly assigning specific tasks to run on specific processors at specific times. The component tasks of such an application require built-in knowledge about the structure of the application and the allocation of resources in order to communicate with other tasks. This coupling of function to structure complicates modification of the application, poses obstacles to runtime changes in the application structure, and prevents reuse of the tasks in other environments. Developers need new tools that allow them to abstract application structure from function.

This document is a user's manual for Durra [Barbacci 91], a language that alleviates the problems described above by allowing an application developer to separate the structural and functional aspects of the application, allowing modification to one without requiring a corresponding modification to the other.

Durra is a task-level application description language.[1] The basic building blocks of the language are the *task description*, which specifies the properties of an associated Ada subprogram, and the *channel description*, which specifies the properties of an Ada package implementing a communication facility. (See Figure 1-1 for a Durra task description template.) Task descriptions may be either *primitive* or *compound*. A primitive task description represents a single thread of control.[2] A compound task description is a composition of other task and channel descriptions. Channel descriptions are syntactically similar to primitive task descriptions although the implementations exhibit different behaviors. Task implementations are active components; they initiate requests to send or receive messages by calling procedures provid-

1. Throughout this document, the term *task* refers to a generalized "thread of control" concept rather than to the analogous Ada construct, except where noted.

2. The actual Ada code that implements a Durra task may, in fact, be a multitasking program. However, from the Durra perspective the program is a single thread of control.

ed by the runtime environment. Channel implementations are passive components; they wait for and respond to requests from the runtime environment. Task and channel implementations are “black boxes,” i.e., the internal workings of a component are not a consideration in the construction of a Durra application description.

```
task taskname (parameter-list)
       -- Values for the parameters are provided in task selections
       ports
          port-declarations
          -- A description of the input-output interface of the task
       behavior
          specification-list
          -- Labelled formal specifications of the behavior of the task
       attributes
          attribute-value-pairs
          -- A list of additional properties of the task
       components --(for compound tasks only)
          component-declarations
          -- A list of task and channel selections
       structures --(for compound tasks only)
          component-connection-structure
          -- A list of component connections
       reconfigurations --(for compound tasks only)
          condition-transition-pairs
          -- A list of conditional structure changes
       clusters --(for compound tasks only)
          cluster-component-associations
          -- A list of named physical groupings of components
end taskname;
```

Figure 1-1: A Template for Task Descriptions

A Durra programmer describes an application as a collection of *processes* (instances of Durra task descriptions) connected to each other in a graph structure via *links* (instances of Durra channel descriptions). Lower level components are used as building blocks for higher level task descriptions. Application descriptions are simply compound task descriptions that describe a complete application.

A detailed specification of the language syntax and semantics can be found in the Durra Reference Manual [Barbacci 91]. We will describe the major concepts of the language in more detail later in this document, but in the next section we step back from consideration of language details in order to consider the larger picture of the Durra application development scenario. In Section 3 we describe the UNIX environment that must be established prior to the start of application development. Section 4 describes the Durra tools and libraries that are available to the application developer. In Section 5 we describe an example distributed application that will be used to illustrate the Durra application development process. Section 6 provides a brief introduction to the Durra description language and example descriptions corresponding to our example. In Sections 7 and 8 we describe how implementations of Durra task

and channel components are developed. Again we provide some simple implementations based on the components of our example application. In Section 9 we describe the *configuration file*, Durra's method for establishing a mapping of the application to physical processing resources. Section 10 explains how to compile and link a Durra application once the Durra descriptions and Ada implementations have been written. Finally, in Section 11 we show how to execute a Durra application.

Throughout this document we will use the following definitions:

- *channel description* – a template written in Durra specifying the properties of a channel implementing a communication facility. Channel descriptions are compiled and stored in Durra libraries.
- *channel implementation* – code written to implement a specific communication facility for application tasks. Channels are passive components that react to requests from tasks. Channel implementations are compiled and stored in object code libraries.
- *cluster* – a group of tasks and channels linked with Durra runtime library code and executed as a multi-threaded program. The source code for the cluster-specific part of a cluster is generated by the Durra compiler. Clusters are compiled and stored in object code libraries.
- *link* – an instance of a channel providing communications between two or more processes.
- *port* – a logical input or output device of a process or link. Input ports get messages from other components; output ports send messages to other components.
- *task description* – a template written in Durra specifying the properties of a task implementing a piece of an application. Task descriptions are compiled and stored in Durra libraries.
- *task implementation* – code written to implement a piece of an application. Tasks are active components that generate requests to the channels. Task implementations are compiled and stored in object code libraries
- *process* – an instance of a task implementing part of an application. A process can execute as single program (actually, a one-thread cluster) or as a thread within a multiple-thread cluster.

2 A Durra Development Scenario

We see three distinct activities in the process of developing a distributed application using Durra: (a) the development of components (task/channel descriptions and implementations), (b) the development of an application description, and (c) the execution of the application. Figure 2-1 illustrates this scenario:

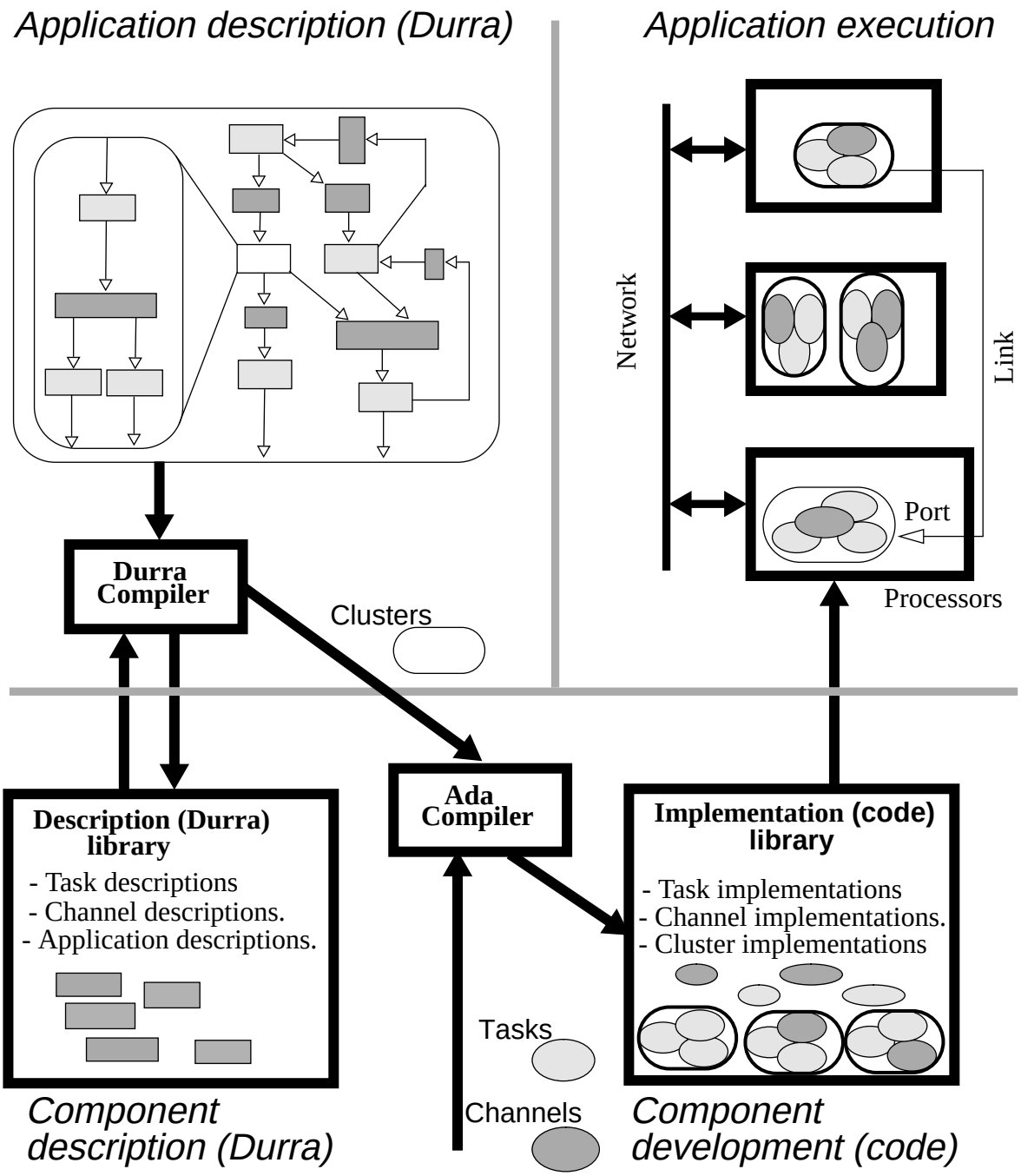


Figure 2-1 Durra Application Development Scenario

Although we will describe the activities in this order, the actual development process might involve successive iterations over these steps. That is, an application is not necessarily developed bottom-up as the scenario might seem to imply. The developer could start with some gross level decomposition of the application into subsystems, and a prototype of the application might be developed with these high level components. Over time, these subsystem might be further decomposed into finer components and the steps repeated as appropriate. The scenario illustrated in the figure is just an illustration of the kinds of activities involved and is not a fixed prescription.

2.1 Component Creation Activities

In this phase, the developer defines the application components (tasks and channels) using domain specific knowledge. Some components might be domain specific, such as sensor processing, map database management, route planning, etc. Other components might be of a more general nature, such as sorting, array operations, etc. An application component consists of a description and an implementation.

A component implementation is an Ada subprogram (in the case of a Durra task) or package (in the case of a Durra channel.) For a given task, there may be many possible implementations, differing in processor type (e.g., Motorola 68020, DEC VAX), performance characteristics, or other attributes. The writing of a task implementation is more or less independent of Durra and involves the coding, debugging, and unit testing of program units on various machines. Available component implementations are stored in the appropriate object code libraries.

A component description is a template specifying properties of a component implementation: the types of data it produces or consumes, the ports it uses to communicate with other tasks, formal specifications of behavior, and other attributes of the implementation.

2.2 Application Creation Activities

In this phase the developer writes an application description that specifies the desired components and their interconnection. Syntactically, an application description is similar to a task description and can be stored in the library as a new component task. This allows the developer to write hierarchical application descriptions.

When the application description is compiled, the Durra compiler identifies clusters of library task and channel descriptions that meet selection criteria specified by the user and generates for each cluster an Ada package body (named *Tables*) containing data and operations specific to that cluster. Only these generated package bodies need to be recompiled if the application description changes. The component implementations are retrieved from the object code library and linked with the Durra runtime library to create executable programs. The main units of these programs constitute an additional thread of control within each cluster (i.e., they constitute an additional component process in the cluster, albeit not one specified by the applica-

tion developer). These additional tasks will be referred to as the *cluster managers.* The application developer can specify how component tasks and channels are to be grouped into clusters. The extreme cases are:

- all components are linked together as one cluster, and
- each component is a separate cluster.

The application developer can also specify what clusters run on each processor.

2.3 Application Execution Activities

To execute an application, the developer loads and starts the clusters in the appropriate processors. The extreme cases are

- all clusters execute on one processor, and
- each cluster executes on a separate processor.

Each cluster's *Tables* package contains information about the structure of the other clusters and the reconfigurations specified by the application developer. This information is used by the cluster manager to provide communication support for the local application processes.

Application processes communicate with each other through the same interface (procedure calls to the cluster manager) regardless of process location (i.e., processes within the same cluster, processes in clusters within the same processor, or processes in clusters in different processors.) The cluster manager implements the port operations (a subset of the interface calls) either as local rendezvous or remote procedure calls depending on the location of the communicating processes.

3 Getting Started with Durra

Before one can begin developing Durra applications, it is necessary to set up the Durra environment. In this section we describe steps that must be taken to configure the Durra system in a UNIX environment, and we introduce the user to the system directory structure and file naming conventions used by Durra tools and libraries.

3.1 Environment Variables

The following environment variables should be defined in the environments of all Durra users:

ADA Defines the root directory of the Ada compilation system being used.

DALL_VERSION Defines the command name to be used by Durra makefiles when the **dall** shell script (see Section 4.2) is invoked.

DURRA_LOG_DIR Defines the directory to which logging information will be written when runtime logging is requested. See Section 11.

DURRA_ROOT Defines the root directory of the Durra system hierarchy. See Section 3.3.

DURRA_RTE Defines the root directory of the Durra runtime library. See Section 3.3.

3.2 Internet Service Definition Requirements

The Durra system requires the inclusion of a set of Internet service definitions in the */etc/services* database. One of the services to be defined is the **Durra_Launcher** (see Section 4.8). Its definition should be of the form:

Durra_Launcher *port-number*/udp

The remaining service definitions are for use by Durra clusters at runtime. Each cluster service definition should be of the form:

Durra_Cluster_*n* *port-number*/tcp

where *n* ranges from 1 to the maximum number of Durra clusters allowed by the Durra runtime (20 by default, though this is modifiable).

The *port-number* for each service should be unique but otherwise is left to the discretion of the system administrator.

3.3 Durra System Directory Structure

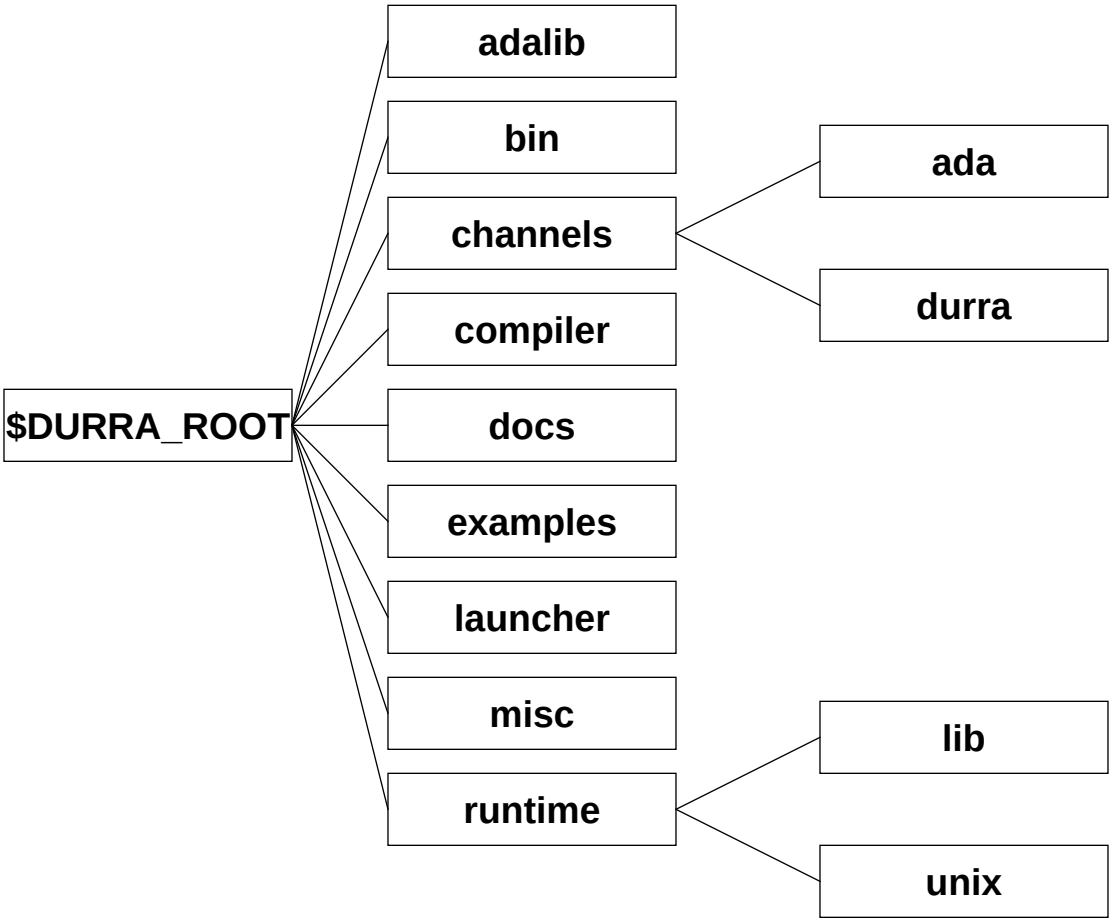


Figure 3-1 The Durra System Directory Structure

Figure 3-1 describes the system hierarchy supplied with the Durra distribution. The top level directory must be identified by the **DURRA_ROOT** environment variable.The subdirectories of this directory contain the following:

- adalib : contains Ada source code implementing various abstract data types and some utility packages.
- bin : contains the Durra tools described in Section 4. This directory should be added to the user's path.
- channels : contains two directories:
  - channels/ada : contains the Ada implementations of the predefined Durra channels
  - channels/durra : contains the Durra descriptions of the predefined Durra channels.

- compiler : contains the Ada source code implementing the Durra compiler.
- docs : contains documentation and reports about the Durra system in PostScript format.
- examples : contains example Durra descriptions which can be used as a learning device and/or a tool for testing your Durra installation.
- launcher : contains the Ada source code implementing the **Durra_Launcher** (see Section 4.8).
- misc : contains the “Cluster_Makefile” template (see Section 4.4) and the channel package templates (see Section 8.1).
- runtime : contains two directories:
  - runtime/lib : contains the Ada source code implementing the portable part of the Durra runtime and the *Durra_Interface* package (see Section 7.1).
  - runtime/unix : contains the UNIX-specific portion of the Ada source code implementing the Durra runtime.

3.4 File Naming Conventions

Over time we have evolved certain file naming conventions that seem to work well for us. The use of these conventions is not mandatory, although failure to observe the file name extension conventions will require minor modifications to the compiler and various utility programs.

Ada source code files supplied with the Durra distribution conform to the convention that files containing specifications end with “**.a**” and files containing bodies end with “**B.a**”.

It is required that each Durra task or channel description or type declaration be in its own file. By convention, the file name will use the Durra object name as its root part and will have the extension “**.durra**”. For example, the task description in Figure 6-1 would be stored in a file called **producer.durra.**

When a Durra source file (i.e., a task or channel description, or a type declaration) is compiled, at least two additional files are generated. One of these is named by the concatenation of the source file name and the further extension “**.TREE**”. Thus, the file generated for the above example would be called **producer.durra.TREE**. This file contains an intermediate representation of the Durra description. The intermediate representation simplifies the use of the description to build more complex Durra descriptions. See Section 6 for more information.

The other file always generated during compilation is named by the concatenation of the source file name and the further extension “**.MAKE**”. Thus, the file generated for the above example would be called **producer.durra.MAKE**. This file contains dependency information which is used by the **dmake** utility (see Section 4.5) when recompiling a Durra description.

If the Durra description being compiled is a complete application description, additional files may be generated. Such a description will include at least one cluster specification, and for each cluster specification the Durra compiler will generate an Ada source file called **Ta-**

blesB.a. Such files will each contain a cluster-specific body for package **Tables**, which is to be linked with Durra runtime library packages to form an executable cluster program. The format of this package body is described in *A Description of Cluster Code Generated by the Durra Compiler* [Doubleday 91].

The **dlibrary** utility (see Section 4.1) is used to create and modify Durra library files. Each of these files is named ".DLIBRARY".

3.5 Descriptive Conventions in This Document

The convention used throughout this document when describing textual commands is that actual text is written in **bold**, and text to be replaced with actual text is written in *italic*. Optional text is {enclosed in braces}. References to Durra specification text or Ada source code text within the running text of the document are written in *italic* as well. In program examples, Ada source and Durra source are written in `Courier`, with Durra reserved words written in **`bold Courier`**.

4 Durra Tools and Runtime Library

In this section we describe in detail the development support tools provided for the Durra application developer and the Durra runtime library, which is linked with developer-provided component implementations to form executable cluster programs.

4.1 The Library Management Utility

The **dlibrary** command implements a modest library management utility:

dlibrary { *options* } { *file-or-directory-name* }

The Durra library is a text file containing information about the Durra compilation units stored in the library and pointers to other libraries containing additional units. When the compiler searches for previously compiled descriptions, it looks first in the current library and then in the libraries referenced in the current library, and so on (in depth-first fashion). The library file is always named **.DLIBRARY** and hence there can be at most one library file per UNIX directory. Compilation units (or symbolic links to them) stored in a library must always reside in the same UNIX directory as the **.DLIBRARY** file.

dlibrary -c

When used with the **-c** option, **dlibrary** creates a new library (or reinitializes an existing one). This command is normally used when starting the development of a new application or library of reusable components.

dlibrary -a *directory-name*

When used with the **-a** option, **dlibrary** extends the library by adding a pointer to another directory to be searched for imported component descriptions or type declarations.

dlibrary -r *directory-name*

This is the complement of the **-a** option. When used with the **-r** option, **dlibrary** removes a pointer to another directory. Component descriptions and type declarations from the library file in that directory are no longer accessible from this library.

dlibrary -d *Durra-source-file-name*

When used with the **-d** option, **dlibrary** deletes a component description or type declaration from the library (the source file is not disturbed.) Normally there is no need to delete library entries using this option because the compiler takes care of insertion and deletion of component descriptions and declarations.

4.2 The Compilation Utility

The **dall** command is a convenience shell script utility which invokes the Durra compiler (see section 4.3) to process a file or group of files:

dall { *compiler-options* } *file-1* {*file-2 file-3 ... file-n* }

All compiler options are passed through to the compiler unchanged. If multiple Durra source files are specified, then any compiler options specified apply to the compilation of each of the files. The **dall** command permits the user to specify only the root of each Durra source file name, omitting the **.durra** extension, which will be added automatically by **dall**.

The Durra compiler writes the compiled form of each source file to its standard output. The **dall** command is responsible for directing the standard output of the compilation to the appropriate ".TREE" file (see Section 3.4) and for updating the Durra library with the new Durra component. For this reason, the Durra compiler is almost always invoked indirectly via the **dall** command.

4.3 The Compiler

The Durra compiler may be invoked directly via the **durra** command:

durra { *options* } *Durra-source-file-name*

The Durra-source-file-name is the name of a text file containing Durra source code. By convention, the file name should have the ".durra" extension. The root name of the file may be different from the name of the Durra component. There are at least two outputs of any successful Durra compilation:

- the intermediate description of the Durra component, which is written to the standard output of the compiler (but see Section 4.2), and
- a text file, for use with **make(1)**, which is used for version control of the Durra application. The file is given the name *"Durra-source-file*.MAKE". For details, see Section 10.

durra -c *configuration-file-name*

The **-c** switch to the compiler specifies a configuration file (see Section 9) to be used when compiling the Durra source file. If no configuration file is specified at compile-time, the compiler warns the user, but it is not an error. The configuration file name must immediately follow the switch with no intervening space.

durra -g

The **-g** switch requests Ada code generation. It indicates that the compiler should generate one *Tables* package body for each cluster in the application description. It is an error to specify this switch when compiling a description that is not a complete application description.

durra -r *cluster-root-directory*

The **-r** switch specifies a directory that will be the root directory for the generated *Tables* package body for each cluster in an application description. The directory name must immediately follow the switch with no intervening space. When the compiler generates cluster code, it creates a directory for each *Tables* package body. The directory name is the name of the cluster. Each of these directories is created as a subdirectory of the directory specified with this switch. If code generation is requested and no *cluster-root-directory* is specified, then the Durra compiler uses the current working directory as the root.

durra -m

The **-m** switch is used to request that source code lines with multiple errors be marked with multiple error messages. The default is to print at most one error message per source code line.

durra -f -G -l -n -p -s -t

These switches are documented here for completeness, though they are not typically needed by users of the Durra compiler. They are for activating traces used to debug various parts of the compiler itself. Compiler bug fixers use **-f** to debug the lexer, **-G** to debug the code generator, **-l** to debug the library reader, **-n** to debug the symbol table processor, **-p** to debug the parser, **-s** to debug the semantic phase, and **-t** to debug the intermediate language tree processor.

4.4 The Cluster Directory Population Utility

The **dmklib** command is a shell script utility used to populate the directories created by the Durra compiler during cluster generation with additional files required for compilation of the clusters:

dmklib { **-m** *Makefile-name* }

The Durra compiler creates subdirectories to hold each of the *Tables* package bodies it has created for a particular application. Each of these directories needs a Makefile for use by the **make(1)** command as part of the framework for maintaining up to date versions of the clusters. Since these Makefiles are almost identical for every cluster, we supply a canonical *Cluster_Makefile* (in directory “$DURRA_ROOT/misc”) which can be modified to the needs of individual applications. Where indicated and necessary, the user should create a new version of the “Cluster_Makefile,” add to the Ada path the names of any Ada libraries which will be needed for compilation of this application, and save the new makefile. With the **-m** switch, the user specifies the name of this makefile. The **dmklib** script copies the Makefile into each of the cluster directories. If the switch is omitted, **dmklib** copies the file named “Makefile” in the current working directory. Note that **dmklib** should be used with care, since it assumes that every subdirectory in the current working directory is a cluster directory. The best practice is

always to name a cluster root directory when compiling an application description, and then to put no other subdirectories under the cluster root directory.

In the SEI environment, the cluster directories need additional subdirectories, the names of which are based on the host architecture, to allow for compilation for different host environments. **dmklib** currently creates subdirectories named ".sun4" and ".sun3" under each cluster directory. These subdirectories contain separate Ada libraries for each supported architecture. Users should modify **dmklib** to create only the subdirectories needed for their environments. No subdirectories are needed at all if only one host architecture is supported.

4.5 The Application Recompilation Utility

The **dmake** shell script utility invokes the **make(1)** command to keep Durra applications up to date:

dmake *Durra-source-file-name-root* { *target* }

The **dmake** command takes as its argument the name of a Durra source file. The ".durra" extension will be appended to the name if it is not explicitly specified. **dmake** assures that the latest version of the file and of all dependent files have been compiled. This utility is strictly for convenience and does nothing that could not be done directly with the **make(1)** command. The two commands below are equivalent:

dmake example world
make -f example.durra.MAKE world

The optional *target* parameter is simply passed through to the **make(1)** command. The target name must correspond to one of the target names in the ".MAKE" file. Since all Durra makefiles have similar formats, there is a standard set of these target names (see Section 10).

4.6 The Runtime Library

The Durra runtime library is supplied with the Durra distribution in directory identified by the environment variable $DURRA_RTE. It depends on additional Ada source code (abstract data type implementations) in directory $DURRA_ROOT/adalib. The runtime library (which includes the cluster main unit) is linked with each compiler-generated *Tables* package body to create a cluster executable program.

The *Durra_Interface* package is included in the runtime library. This is the application programmer's interface to the Durra runtime. It is described in detail in Section 7.1.

4.7 The Predefined Channel Library

A collection of predefined channel specifications and implementations are included with the Durra distribution. The Durra specifications of the channels are in "$DURRA_ROOT/channels/durra" and the Ada implementations are in "$DURRA_ROOT/channels/ada."

As of this writing, the following channel implementations are provided:

- *FIFO_Channel*

  This channel implements a FIFO communication buffer. It allows for one input port and one output port.

- *Merge_FIFO*

  This channel implements a FIFO-merge communication buffer. A message arriving at any of the input ports is sent to all output ports in FIFO order. It allows for up to 10 input ports and one output port. The maximum number of ports is easily modified in the specification of the package.

- *Broadcast*

  This channel implements a broadcast communication buffer. A message arriving at the input port is sent to all output ports in FIFO order. It allows for up to 10 output ports and one input port. The maximum number of input ports is easily modified in the specification of the package.

- *Merge_Broadcast*

  This channel implements a merge-broadcast communication buffer. A message arriving at any of the input ports is sent to all output ports in FIFO order. It allows for up to 10 input ports and up to 10 output ports. The maximum number of ports is easily modified in the specification of the package.

- *Exclusive_Merge_Broadcast*

  This channel implements an exclusive merge-broadcast communication buffer. A message arriving at the *i*th input port is sent to all except the *i*th output ports in FIFO order. It allows for up to 10 input ports and up to 10 output ports. The maximum number of ports is easily modified in the specification of the package.

Although it is simple to expand the maximum number of ports a channel may handle, care should be exercised that the maximum not be made unduly large. There is a performance penalty associated with a larger number of potential ports, since each potential port must be associated with an Ada task entry call, even if the port is not used.

4.8 The Cluster Activation Daemon (Durra_Launcher)

The Durra_Launcher is a daemon process which should be running on any processor that is to participate in the execution of a Durra application. When the master cluster is started from the shell by the user, it contacts the Durra_Launcher on each host where a subordinate cluster is expected to run and tells it to start the cluster. The master informs the Durra_Launcher of the location of each executable image that it is to start. If the user wishes to avoid using the Durra_Launcher, there is a command line argument (see Section 11) which can be passed to the master cluster which tells it not to try to launch the other clusters. In that case, each cluster must be started by hand.

5 Example : An Informal Durra Application Construction Process

In this section and following ones we describe the process of developing a Durra application, including:

- the specification in Durra of the components of the application (Section 6),
- the coding of the Ada implementations of those components (Sections 7 and 8),
- the use of a configuration file to map clusters to physical processor resources (Section 9),
- the compilation and linking of the Durra and Ada elements into executable Ada programs called clusters (Section 10), and
- the execution of the application (Section 11).

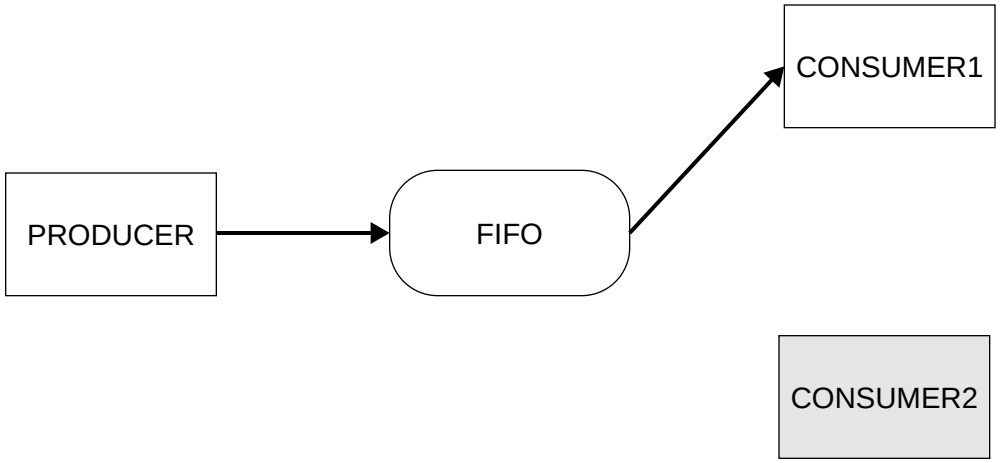
This manual is primarily intended as a guide to use of the Durra system and not as a tutorial on distributed system requirements specification and design. Therefore, we assume for our purposes the existence of a set of requirements and a high-level design derived from those requirements. The abstract components of the system and the dataflow paths will already have been identified in this design, and we can concentrate on expressing this abstraction in the form of a Durra application description.

5.1 The Example Application

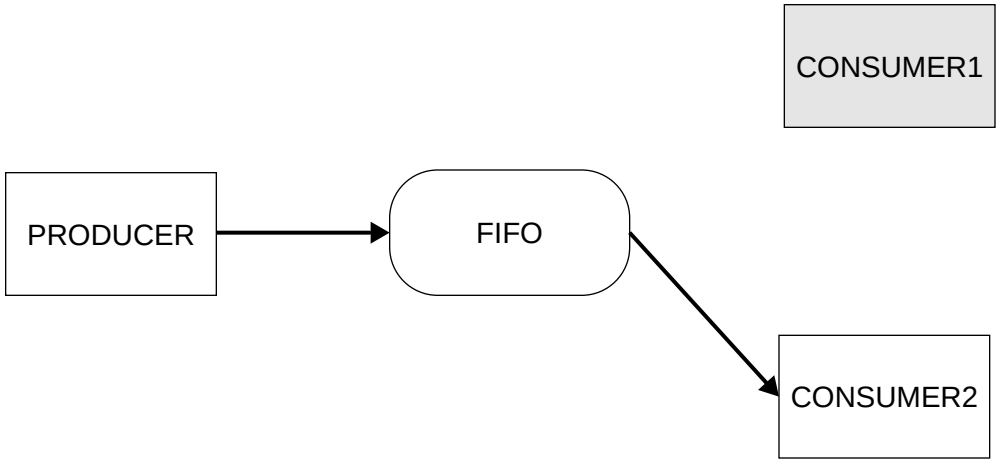
We have chosen a dynamic producer-consumer application to illustrate Durra development since this application is a simple one which can be described quite easily and yet demonstrates the most important features of the Durra system.

The example application consists of a task that produces messages, two tasks that consume messages, and a channel that relays messages from the producer to a consumer in FIFO order. These components operate in two distinct runtime modes, or *configurations* (see Figure 5-1). In the initial configuration, the producer sends messages to the first consumer until the first consumer indicates to the Durra runtime that it has consumed all the messages it requires. The second consumer is inactive during this configuration. The Durra runtime then reconfigures the application so that the second consumer begins receiving the messages and the first

consumer is made inactive. When the second consumer has finished, the application terminates.



a) Initial Configuration



b) Second Configuration

Figure 5-1 Dynamic Producer-Consumer Application Structure

6 Writing a Durra Application Description

In this section we give more detail about the semantics of the various parts of a Durra component description and show how Durra allows the programmer to specify the structure, both static and dynamic, of a distributed application. During the following discussion, refer to the task description template presented in Figure 1-1.

6.1 Type Declarations

Durra data type declarations define either a *size* type or a *union* type. A size type declaration associates an identifier with a data size (or a range of data sizes) expressed in bits. A union type declaration defines a new type as the union of one or more previously declared types. This type concept is analogous to our “black box” treatment of tasks—no semantic information other than the type name and the size of the data is derived from a type declaration. Here are some examples of Durra type declarations:

```
type byte is size 8;
type scalar is size 4*sizeof(byte);
type message is union (byte,scalar);
```

6.2 Port Specifications

A component’s input/output interface is specified by the *ports* section of its description. Ports are named, unidirectional, locally-defined conduits through which processes may transmit and receive data. Ports have a Durra data type associated with them to allow semantic checking of intercomponent port connections.

6.3 Behavioral Specifications

The *behavior* section of the component description includes zero or more formal specifications of the behavior of the component’s actual implementation. These specifications are not interpreted by the Durra compiler directly, but by associated tools. Although behavioral specifications are not part of the Durra language, a Durra component description provides a convenient placeholder for such specifications. Component descriptions containing behavioral specifications may then be used as components of an application description. A specification analysis tool is thus provided with a framework for reasoning about the composition of the specifications within an application architecture.

6.4 Attribute Specifications

The *attributes* section defines additional properties of the component, such as version number or type of processor required. A primitive task description must be associated (via an attribute value) with a specific Ada procedure that is its implementation.

6.5 Primitive Component Descriptions

Figure 6-1 is an example of a primitive Durra task description. The task *producer* has one output port for data of type *message*. It is intended to run on a Sun4 processor, and its implementation is the Ada procedure “producer” in the library “/usr/durra/srclib”.

```
task producer
  ports
    output : out message;
  attributes
    processor = “sun4”;
    procedure_name = “producer”;
    library = “/usr/durra/srclib”;
end producer;
```

Figure 6-1: A Primitive Durra Task Description

A channel description is always primitive and is associated with a specific Ada package that implements it. Channels are intermediary processes which control the flow of data between user processes. Channel descriptions and implementations for many frequently used communication disciplines are provided as part of the Durra support environment. These include FIFO and priority queue, broadcast, and merge, among others. Additional channel implementations may be provided by application developers to suit specialized application needs. Such developer-provided implementations must be written with care, however, since channels are required to present a uniform interface to the Durra runtime and to follow established Durra runtime practice in storage management and blocking behavior. See Section 8 for details.

Both task and channel descriptions may be parameterized to allow for more flexible use of components. For example, one instance of a broadcast channel may be defined to have 3 output ports and another instance to have 10 output ports. Figure 6-2 contains descriptions of a generic channel (*fifo*) and a generic task (*consumer*). Each has a formal parameter that determines the data type of messages it can handle. The *buffer_size* parameter for the *fifo* channel specifies the number of messages that can be buffered by each input port of the channel. Parameter values are supplied by *task/channel selections.* Selections are templates (identical to primitive description templates) that are used in compound task descriptions to select lower level components with the desired properties.

6.6 Compound Task Descriptions

A compound task description must include additional information about its structure. Its component processes and links are defined in its *components* section and the manner in which they are logically connected (which may vary dynamically) is specified in its *structures* section. If the structure of the compound task is allowed to vary, then there must be a *reconfigurations* section that describes a set of structural changes and the conditions under which the changes will occur. The *clusters* section specifies the physical grouping of components into executable

```
channel fifo(msg_type: identifier,buffer_size: integer)
  ports
    input: in msg_type;
    output: out msg_type;
  attributes
    processor = “sun4”;
    bound = buffer_size;
    package_name = “fifo_channel”;
    library = “/usr/durra/channels”;
end fifo;

task consumer (msg_type:identifier)
  ports
    input: in msg_type;
  attributes
    processor = “sun4”;
    procedure_name = “consumer”;
    library = “/usr/durra/srclib”;
end consumer;
```

Figure 6-2: Generic Channel and Task Descriptions

images, which may well be orthogonal to the logical connections described in the *structures* section.

In Figure 6-3, we provide a Durra description of the classic producer-consumer problem as an example of a compound task description which also happens to be an application description. The building blocks for the task *producer_consumer* are the primitive components identified in Figure 6-1 and Figure 6-2. The *c* and *buffer* declarations in the *components* section are exam-

```
task producer_consumer
 components
   p: task producer;
   c: task consumer(message);
   buffer: channel fifo(message,10);
 structure
   L1:  begin
          baseline p, c, buffer;
            buffer: p.output >> c.input;
        end L1;
 clusters
   cl1 : p, buffer;
   cl2 : c;
end producer_consumer;
```

Figure 6-3: A Producer-Consumer Application Description

ples of component selections that supply arguments to bind values to the formal parameters of the selected descriptions. The *structure* section in Figure 6-3 is very simple. In Durra, the structure of an application is described as a collection of labelled *configuration levels*, which may be either nested or independent. There is only one configuration level (*L1*) in this application description. The *baseline* statement defines which processes and links are active at a given level. Intercomponent connections are expressed in terms of the link implementing the connection. Thus, the link *buffer* connects the port *p.output* to the port *c.input*. The Durra compiler ensures that all ports are connected and that they are connected to ports of the proper data type and direction.

The Durra programmer specifies the distribution of application components by assigning them to virtual nodes called *clusters*. The *clusters* section of the description specifies that process *p* and link *buffer* will be physically grouped together at runtime, but process *c* will be linked into a separate executable program. This concept will be discussed in more detail in Section 10.

We require a more complex application description in order to demonstrate Durra's ability to express dynamic reconfiguration requirements.Figure 6-4 is an extension of the description in

```
task dynamic_producer_consumer
 components
  p: task producer;
  c1, c2: task consumer(message);
  buffer: channel fifo(message, 10);
 structure
  L1: begin
        baseline p, c1, buffer;
        buffer: p.output >> c1.input;
    L2: begin
          include c2;
          exclude c1;
          buffer: p.output >> c2.input;
        end L2;
       end L1;
 reconfigurations
  enter => L1;
  L1 => L2 when signal(c1, 1);
  L2 => exit when finish(c2);
 clusters
  cl1 : p, buffer;
  cl2 : c1, c2;
end dynamic_producer_consumer;
```

Figure 6-4: A Producer-Consumer Application Description Featuring Dynamic Reconfiguration

Figure 6-3. One new component, a second instance of the *consumer* task, has been added. The *structures* section in this example has been expanded to include a second configuration

level, *L2*, which is nested within level *L1*. This level incorporates the new component, *c2*, and excludes a component from *L1*, *c1*. Since process *p* and link *buffer* are not explicitly excluded from the nested configuration description, they survive into the new configuration. In the new configuration, port *c1.input* is disconnected and port *c2.input* is now connected to the output of *buffer*. This structural specification corresponds to the application structures shown graphically in Figure 5-1. It is helpful to think of the *structures* section in an application description as a variant data record which describes the static structure of the application at each configuration level. The structures section implies nothing about the transition between levels; that is the purpose of the *reconfigurations* section.

The *reconfigurations* section of the *dynamic_producer_consumer* application description prescribes the conditions under which the configurations specified in the *structures* section shall be entered. Transition from one configuration to another is indicated by a configuration name pair on opposite sides of an arrow operator. When the application is in the configuration on the left-hand side of the arrow, the application is eligible to reconfigure to the configuration on the right-hand side of the arrow. A condition is usually associated with the transition, as in the transition from *L1* to *L2* in Figure 6-4. In this particular case, the transition will occur when the Durra runtime receives a *signal* (see Section 7.1.10) from process *c*1. Durra assigns no semantic content to particular signal values; the interpretation of such signals is a function of the application description. The transition to configuration *L1* is a special case—the keyword **enter** indicates that *L1* is to be entered unconditionally at application start-up. The transition to the **exit** configuration from *L2* indicates that this application will terminate when process *c2* has called the *Finish* operation provided by the Durra runtime (see Section 7.1.4).

7 Writing a Durra Task Implementation

In this section we describe the Durra services available to an Ada program through the application programmer's interface (API) to the Durra runtime. We then show how to use those services to write implementations of the *producer* and *consumer* tasks used in our example application.

Note that for purposes of exposition we have shown the development of all Durra component descriptions before the development of the component implementations. This is not intended to imply that this is the only or even the preferred sequence of activities. It is perfectly reasonable to develop all implementations first or to interleave component implementation and description development activities.

7.1 The Durra Application Programmer's Interface

The Durra runtime library provides an Ada package called *Durra_Interface*. This package must be "withed" by all Durra component implementations. It is the API to Durra runtime services. In this section we will describe the services provided by *Durra_Interface*. For a complete listing of the package specification, see Appendix B.

7.1.1 Types and Constants

```
type Type_ID        is private;
type Input_Port_ID  is private;
type Output_Port_ID is private;
type Process_ID     is private;

subtype Signal_Range is DT.Signal_Range;
subtype Message_Priority is DT.Message_Priority_Range;

NULL_TYPE_ID          : constant Type_ID;
NULL_MESSAGE_PRIORITY : constant Message_Priority;
```

Durra types, ports, and processes have runtime handles that uniquely identify them. The private type declarations shown above are the types that represent those handles. The subtype *Signal_Range* defines the range that application processes may use for signals to the Durra runtime (currently defined as the range of the predefined Ada type *Integer*.) The subtype Message_Priority defines the range of priorities that application processes may assign to individual messages as they are sent (currently defined as the range of the predefined Ada type *Natural*.) The constants above define distinguished null values for the types *Type_ID* and *Message_Priority*.

7.1.2 Exceptions

```
Bad_Port_Name      : exception;
Bad_Port_ID        : exception;
Bad_Process_ID     : exception;
Bad_Type_Name      : exception;
Bad_Type_ID        : exception;
Uninitialized      : exception;
Already_Initialized: exception;
```

The exceptions identified above are exported by *Durra_Interface*. The first five of these are raised when invalid parameters are passed to *Durra_Interface* operations. *Already_Initialized* is raised by the *Init* operation (see Section 7.1.4) when it has already been called by a process. *Uninitialized* is raised when any other operation is called before *Init* has been called.

7.1.3 Time Query Operations

```
function Get_Application_Time return DURATION;

function Get_Process_Time (Process : in Process_ID) return DURATION;

function Get_Day_Time return DURATION;
```

This group of operations queries the Durra runtime for various time values (expressed as the Ada *Duration* type.) *Get_Application_Time* returns the time elapsed since the start of the application of which the calling process is a component. *Get_Process_Time* returns the time elapsed since the start of the calling process. *Get_Day_Time* returns the time elapsed since midnight of the current day.

7.1.4 *Init* and *Finish* Operations

```
procedure Init (Task_Sequence_Number : in     POSITIVE;
                Process_Identifier   :    out Process_ID);

procedure Finish (Process : in Process_ID);
```

The *Init* procedure announces the presence of the calling process to the Durra runtime. It must be the first Durra service requested by any Durra process. The *Finish* procedure informs the runtime that the calling process has finished its computation and is terminating. It must be the final Durra service requested by any Durra process.

On calling *Init*, a process must pass as the *Task_Sequence_Number* parameter the positive scalar value it has received from the Durra runtime at process activation time. The *Init* procedure returns a unique *Process_ID* for the calling process in the parameter *Process_Identifier*. This *Process_ID* is required as an input argument by all other Durra operations (except the *Get_Application_TIme* and *Get_Day_Time* operations, which are not process-specific.)

7.1.5 Process Attribute Query Operation

```
function Get_Attribute (Process        : in Process_ID;
                        Attribute_Name : in STRING) return STRING;
```

The *Get_Attribute* function returns the string value associated with the process attribute name specified by the *Attribute_Name* parameter. Attribute names are case-insensitive. If no such attribute has been specified for this process, *Get_Attribute* returns a null string.

7.1.6 Port and Type Identification Query Operations

```
procedure Get_PortId (Process     : in     Process_ID;
                      Port_Name   : in     STRING;
                      Port        :    out Input_Port_ID;
                      Data_Size   :    out NATURAL;
                      Port_Type_ID:    out Type_ID);

procedure Get_PortId (Process     : in     Process_ID;
                      Port_Name   : in     STRING;
                      Port        :    out Output_Port_ID;
                      Data_Size   :    out NATURAL;
                      Port_Type_ID:    out Type_ID);

procedure Get_TypeId (Type_Name   : in     STRING;
                      Data_Type   :    out Type_ID;
                      Type_Size   :    out NATURAL);
```

Just as processes have unique IDs supplied by the Durra runtime, so do Durra ports and types. The *Get_PortId* and *Get_TypeId* operations are used to acquire these identifiers.

The *Get_PortId* operation is overloaded to return distinct identifier types for input ports and output ports. In both versions, the calling process passes in a string, *Port_Name*, which is the same as the name given to a port of the calling process in the Durra description. If no port with the specified name and dataflow direction exists, *Get_PortId* raises the *Bad_Port_Name* exception. Otherwise, it returns the correct type of *Port_ID* in parameter *Port* and the *Type_ID* of the Durra type associated with the port in parameter *Port_Type_ID*. It also returns the maximum size (in bytes) of data allowed to pass through the port in parameter *Data_Size*. If the data has no fixed maximum size (i.e., it is a variable length data type), then by convention the value of *Data_Size* on return will be 0.

When calling the *Get_TypeId* operation, the process passes in a string, *Type_Name*, which is the same as the name given in a type declaration in the Durra application description. If no type with the specified name exists, *Get_TypeId* raises the *Bad_Type_Name* exception.Otherwise, it returns the *Type_ID* in parameter *Data_Type*. It also returns the maximum size (in bytes) of objects of the type in parameter *Type_Size*. If objects of the type have no fixed maximum size (i.e., it is a variable length data type), then by convention the value of *Type_Size* on return will be 0.

Port_IDs are only unique within the context of the process, so it is an error to obtain a local *Port_ID* and pass that identifier to another process.

7.1.7 Port Status Query Operations

```
procedure Test_Input_Port
            (Process            : in     Process_ID;
             Port               : in     Input_Port_ID;
             Type_of_Next_Input :    out Type_ID;
             Size_of_Next_Input :    out NATURAL;
             Inputs_Available   :    out NATURAL);

procedure Test_Output_Port (Process          : in     Process_ID;
                            Port             : in     Output_Port_ID;
                            Spaces_Available :    out NATURAL);
```

Sometimes a Durra process will need to query the states of its local ports. For that reason the Durra API provides the *Test_Input_Port* and *Test_Output_Port* operations.

For a given *Input_Port_ID* in parameter *Port*, *Test_Input_Port* returns the number of messages currently available to be read at that port in the parameter *Inputs_Available*. If *Inputs_Available* is nonzero, then the procedure also returns the *Type_ID* of the next available message in *Type_of_Next_Input* and the size (in bytes) of the next available message in *Size_of_Next_Input*. Otherwise, both parameters will have the value 0 upon return.

The *Test_Input_Port* operation can be useful when the process needs to attempt input only when it can be assured that it will not block. If *Test_Input_Port* returns a positive value in *Inputs_Available*, then at least that many input operations can be done on that port without fear of blocking. *Test_Input_Port* is also useful if the data passing through the port are objects of a variable size type or a union type. In such a case the process may need to know the base type or the actual size of the data in advance of any attempted input operation in order to know which buffer to use for data storage.

For a given *Output_Port_ID* in parameter *Port*, *Test_Output_Port* returns in the parameter *Spaces_Available* the minimum number of messages that can be sent to that port before further output operations may cause the process to block.

For example, a process may be written so that it alternates a sequence of output operations with other processing. The programmer does not wish the process to block while trying to send output messages. The process can use the *Test_Output_Port* operation to determine how many output operations can safely be done, then send that many messages, and then perform the other processing.

7.1.8 Input Operations

```
procedure Get_Port (Process       : in     Process_ID;
                    Port          : in     Input_Port_ID;
                    Data          : in     System.Address;
                    Data_Size     :    out NATURAL;
                    Data_Type     :    out Type_ID);

procedure Get_Port (Process   : in     Process_ID;
                    Port      : in     Input_Port_ID;
                    Data      : in     System.Address;
                    Data_Size :    out NATURAL;
                    Data_Type :    out Type_ID;
                    Got_Data  :    out BOOLEAN);
```

The two overloaded *Get_Port* operations provide the message reception capability in Durra. A call to the first version will cause the calling process to block for potentially unbounded time if no message is available at the specified port. A call to the second version will never cause the process to block for potentially unbounded time; there may be a brief blocking overhead related to process scheduling or remote communication.

In both versions, the calling process supplies an *Input_Port_ID* in parameter *Port* and the address of a buffer where the received message should be stored in parameter *Data*. In the blocking version, when *Get_Port* returns the parameters *Data_Size* and *Data_Type* will contain the size (in bytes) and the Durra *Type_ID*, respectively, of the message received. In the non-blocking version, if *Get_Port* returns with the value *True* in the parameter *Got_Data*, then *Data_Size* and *Data_Type* will contain the same information as in the blocking version. If *Get_Port* returns with value *False* in parameter *Got_Data*, then there was no data currently available and parameters *Data_Size* and *Data_Type* will contain values 0 and *Null_Type_ID*, respectively.

Note that non-blocking reception of messages can be accomplished using either the non-blocking *Get_Port* operation or a combination of *Test_Input_Port* and the blocking *Get_Port* operation. The latter will likely be slightly more efficient in the case where messages arrive infrequently. The former will likely be much more efficient when messages are expected to be available almost all the time.

7.1.9 Output Operations

```
procedure Send_Port (Process   : in Process_ID;
                     Port      : in Output_Port_ID;
                     Data      : in System.Address;
                     Data_Size : in NATURAL;
                     Data_Type : in Type_ID := NULL_TYPE_ID;
                     Priority  : in Message_Priority
                                    := NULL_MESSAGE_PRIORITY);
```

```
procedure Send_Port (Process   : in     Process_ID;
                     Port      : in     Output_Port_ID;
                     Data      : in     System.Address;
                     Data_Size : in     NATURAL;
                     Data_Sent :    out BOOLEAN;
                     Data_Type : in     Type_ID := NULL_TYPE_ID;
                     Priority  : in     Message_Priority
                                           := NULL_MESSAGE_PRIORITY);
```

The two overloaded *Send_Port* operations provide the message sending capability in Durra. A call to the first version will cause the calling process to block for potentially unbounded time if no space is available in the buffer associated with the specified port. A call to the second version will never cause the process to block for potentially unbounded time; there may be a brief blocking overhead related to process scheduling or remote communication.

In both versions, the calling process supplies an *Output_Port_ID* in parameter *Port*, the message's sending address in parameter *Data*, the size (in bytes) of the message in parameter *Data_Size*, the Durra *Type_ID* of the message in parameter *Data_Type*, and the priority of the message in parameter *Priority*. In the non-blocking version, when *Send_Port* returns the parameter, *Data_Sent* indicates whether or not the data was actually sent. If it was not sent, then the operation can be retried later.

In either version, the *Data_Type* and *Priority* parameters have default values of *Null_Type_ID* and *Null_Message_Priority*, respectively. It is recommended practice to supply an actual *Type_ID* whenever possible. By convention, the Durra runtime omits type-checking of messages sent with a *Null_Type_ID*. We have found this useful when generating generic task implementations from formal specifications. The programmer sacrifices some type safety and runtime speed (because storage management cannot be optimized) when using this convention.

Message priorities are subordinate to process priorities; that is, the high-priority process wishing to send a low-priority message will get priority over the low-priority process sending a high-priority message. In any event, message priorities will have no effect unless the link associated with the output port respects message priorities. In our UNIX implementation, which utilizes sockets and setup protocol, message priorities are not respected for network communications, either.

Note that non-blocking output of messages can be accomplished using either the non-blocking *Send_Port* operation or a combination of *Test_Output_Port* and the blocking *Send_Port* operation. The latter will likely be slightly more efficient in the case where the output buffer is frequently filled to capacity. The former will likely be much more efficient when the output buffer is expected to be less than full almost all the time.

7.1.10 Reconfiguration Support Operations

```
procedure Raise_Signal (Process       : in Process_ID;
                        Signal_Number : in Signal_Range);

procedure Safe (Process : in Process_ID);
```

The Durra approach to dynamic reconfiguration of the application structure at runtime relies on application cooperation with the runtime. The *Raise_Signal* and *Safe* operations provide the application with the means to support dynamic reconfiguration activities.

A Durra process may use the *Raise_Signal* operation to notify the runtime of some condition. The signal semantics are defined totally by the Durra application description of which the process is a component. For example, a task implementation may contain a call to *Raise_Signal* with a *Signal_Number* value of 1. If the application description defines the signal condition 1 for that process, and if the application is in the configuration where the signal condition definition occurs, then some action (defined by the application description again) will be taken. Otherwise, the signal will be ignored by the Durra runtime.

The *Safe* operation informs the Durra runtime that the process is at a point in its execution where it can be reconfigured safely. If no reconfiguration involving this process has been requested, the call returns immediately. Otherwise, the runtime doesn't allow the call to return to the process until the reconfiguration has completed.

Lacking knowledge of the algorithm being executed by any process, the runtime cannot determine whether or not the process is in the middle of some transaction which must be completed before a reconfiguration occurs. Therefore, task implementations being designed to be safely reconfigurable should make use of the *Safe* operation at strategic points in the code. Failure to do so may result in the runtime performing the reconfiguration after some time-out period, whether the process is ready for reconfiguration or not.

7.2 Example Usage of the *Durra_Interface* Package

In this section we show how *Durra_Interface* services are used in the development of two example Durra task implementations: the producer and consumer tasks from our canonical example.

The implementation of a Durra task must be an Ada procedure with a single formal parameter of predefined type *Positive*. This value, which is used to establish a unique identifier for the Durra process at runtime, is passed to the procedure from the Durra cluster manager. Since each Durra process is a separate thread of control, the cluster source code generated by the Durra compiler uses the subprogram implementing a Durra task as a parameter to a generic package called *Process_Shell*. Each instance of *Process_Shell* contains an Ada task devoted to calling the formal subprogram associated with that instance. The form of the Ada code generated by the Durra compiler is described elsewhere [Doubleday 91].

All Durra task implementations will be similar in some respects. An implementation must call the *Init* operation first in order to get its *Process_ID*. It then uses the *Get_PortID* and *Get_TypeID* operations to get identifiers for the Durra ports and types that it expects to employ. All other Durra services are then available, depending on the communication needs of the particular task. When the task has completed its work, the *Finish* operation must be used to notify the Durra runtime.

7.2.1 The Producer Implementation

The Ada subprogram at the end of this section is one possible implementation of the Durra *producer* task described earlier. To reduce complexity for purposes of this document, this implementation sends only messages of Durra type *Scalar*, even though the Durra description allows sending messages of type *Scalar* or *Byte*. The implementation compares the size of objects of the Durra type *Scalar* to the size of the objects it intends to transmit, insuring no size mismatch. The implementation then loops, alternating algorithmic processing with *Send_Port* and *Safe* operations. *Safe* operations should be liberally included in implementations intended to be dynamically reconfigurable; see the discussion in Section 7.1.10.

The implementation may block when attempting to send its message; this depends on the rate of message production, the rate of consumption, and the buffer space allotted to the link connecting the *producer* and the *consumer* processes. This implementation will send messages indefinitely unless the algorithmic portion of the code modifies the loop condition or some other system component causes a reconfiguration at runtime. In the former case, the implementation terminates by calling the *Finish* operation. In the latter, the implementation will block at the *Safe* call and be reactivated only if it survives into the new configuration. The implementation includes a handler for exceptions that may be raised by Durra operations, as should all implementations.

```
with Durra_Interface; use Durra_Interface;
with Error;

procedure Producer (Task_Number : in Positive) is

  package DI renames Durra_Interface;
  package E  renames Error;

  This_Process      : DI.Process_ID;
    -- The unique identifier of this process.
  Output_Port       : DI.Output_Port_ID;
    -- The unique identifier of the output port of this process.
  Output_Port_Type,
    -- The unique identifier of the type associated with the output port.
  Scalar_Type       : DI.Type_ID;
    -- The unique identifier of the type of message this process will
    -- produce. If Output_Port_Type is non-union, then these two Type_IDs
    -- must be the same.  Otherwise, Scalar_Type must be the same union
    -- type or one of the elements of type Output_Port_Type.
  Output_Data_Size,
    -- The size of data allowed to pass through the output port (zero if
    -- the data is variable-length, as in this case).
```

```
  Scalar_Size       : Natural;
    -- The size data objects of type Scalar_Type.

  Scalar_Buffer     : Natural := 0;
  Some_Condition    : Boolean := True;

begin
  DI.Init (Task_Number, This_Process);
  DI.Get_PortId (
      This_Process,“output”,Output_Port,Output_Data_Size,Output_Port_Type);
  DI.Get_TypeId (“scalar”,Scalar_Type,Scalar_Size);
  if Scalar_Size /= Natural’SIZE/8  then
    E.Warning (“Data type mismatch in process “ &
                DI.Get_Attribute(This_Process, “process_name”));
  else
    while Some_Condition loop
      DI.Safe(This_Process);
      -- <some algorithm resulting in new data in the Scalar_Buffer>
      DI.Send_Port(This_Process,
                   Output_Port,
                   Scalar_Buffer’ADDRESS,
                   Scalar_Size,
                   Scalar_Type);
    end loop;
  end if;

  DI.Finish(This_Process);

exception
  when DI.Bad_Port_Name =>
    E.Warning (“Bad port name in process “ &
                DI.Get_Attribute(This_Process, “process_name”));
  when DI.Bad_Type_Name =>
    E.Warning (“Bad type name in process “ &
                DI.Get_Attribute(This_Process, “process_name”));
  when others =>
    E.Warning (“Unhandled exception raised in process “ &
                DI.Get_Attribute(This_Process, “process_name”));
    raise;
end Producer;
```

7.2.2 The Consumer Implementation

The Ada subprogram at the end of this section is one possible implementation of the Durra *consumer* task described earlier. The implementation compares the size of objects of the Durra types *Scalar* and *Byte* to the size of the objects it expects to receive, insuring no size mismatch. The implementation then loops until 100 messages have been received.

In this example we show one method by which a programmer can be assured that an implementation will not block waiting for communication. Before calling the *Get_Port* operation, the implementation calls the *Test_Input_Port* operation, which determines whether or not an input message is available to be received. If there is no message, then no *Get_Port* is done.

The example also demonstrates that the *Test_Input_Port* operation can be used to determine in advance of receipt the type and size of the next message. This is useful when streams of

mixed message types or variable length objects can be received through a single port, because it allows the consuming task to deposit the data in differing locations according to the type of the data or its storage requirements.

This implementation of the *consumer* task raises a signal to the runtime when it has received all the data it requires. The interpretation of the signal is completely up to the Durra runtime, as directed by the application description. In our example application description, the signal raised will cause a reconfiguration from level *L1* to level *L2*, but at level *L2* the signal is ignored by the runtime, since no reconfiguration is conditioned on it.

```
with Durra_Interface; use Durra_Interface;
with Error;

procedure Consumer (Task_Number : in Positive) is

  package DI renames Durra_Interface;
  package E  renames Error;

  This_Process      : DI.Process_ID;
    -- The unique identifier of this process.
  Input_Port        : DI.Input_Port_ID;
    -- The unique identifier of the input port of this process.

  Input_Port_Type,
    -- The unique identifier of the type associated with the input port.
    -- In this case the type is “message”, which is the union type composed
    -- of the “scalar” and “byte” types.
  Input_Type,
    -- The unique identifier of the type of a received data object.
  Scalar_Type,
    -- The unique identifier of the type “scalar”.
  Byte_Type         : DI.Type_ID;
    -- The unique identifier of the type “byte”.

  Input_Data_Size,
    -- The size of data allowed to pass through the Input port (zero if
    -- the data is variable-length, as in this case).
  Input_Size,
    -- The size of a received data object.
  Scalar_Size,
    -- The size of data objects of type Scalar_Type.
  Byte_Size         : Natural;
    -- The size of data objects of type Byte_Type.

  Scalar_Buffer     : Natural := 0;
  Byte_Buffer       : Character := ASCII.NUL;
  Some_Condition    : Boolean := True;
  Message_Count,
  Inputs_Available  : Natural := 0;

begin
  DI.Init (Task_Number, This_Process);
  DI.Get_PortId (
       This_Process,“input”,Input_Port,Input_Data_Size,Input_Port_Type);
  DI.Get_TypeId (“scalar”,Scalar_Type,Scalar_Size);
  DI.Get_TypeId (“byte”,Byte_Type,Byte_Size);
```

```
  if Scalar_Size /= Natural'SIZE/8 or Byte_Size /= Character'SIZE/8 then
    E.Warning ("Data type mismatch in process " &
                DI.Get_Attribute(This_Process, "process_name"));
  else
    while Message_Count <= 100 loop
      DI.Safe(This_Process);
      DI.Test_Input_Port (
           This_Process,Input_Port,Input_Type,Input_Size,Inputs_Available);
      if Inputs_Available > 0 then
        Message_Count := Message_Count + 1;
        if Input_Type = Scalar_Type then
          DI.Get_Port(This_Process,
                      Input_Port,
                      Scalar_Buffer'ADDRESS,
                      Input_Size,
                      Input_Type);
          -- <some algorithmic processing>
        elsif Input_Type = Byte_Type then
          DI.Get_Port(This_Process,
                      Input_Port,
                      Byte_Buffer'ADDRESS,
                      Input_Size,
                      Input_Type);
          -- <some algorithmic processing>
        else
          E.Warning("Received unexpected message type");
          exit;
        end if;
      end if;
    end loop;
  end if;

  DI.Raise_Signal(This_Process, 1);
  DI.Finish(This_Process);

exception
  when DI.Bad_Port_Name =>
    E.Warning ("Bad port name in process " &
                DI.Get_Attribute(This_Process, "process_name"));
  when DI.Bad_Type_Name =>
    E.Warning ("Bad type name in process " &
                DI.Get_Attribute(This_Process, "process_name"));
  when others =>
    E.Warning ("Unhandled exception raised in process " &
                DI.Get_Attribute(This_Process, "process_name"));
    raise;
end Consumer;
```

8 Writing a Durra Channel Implementation

Each channel in a Durra application description is implemented by an Ada package. The package defines a task type, from which task objects are declared for each link in the Durra description. Durra descriptions and Ada implementations of a number of commonly used channels are provided with the Durra distribution. Programmers are allowed to create additional channel descriptions and implementations. However, unlike a task implementation, the implementation of a channel requires knowledge of and access to the internals of the Durra runtime. Development of new channels is therefore more error-prone than development of new tasks, and should be reserved for the most knowledgable programmers. In this section, we will identify rules that must be followed when developing channel implementations in order to ensure correct behavior of the runtime.

To assist in the development of new channel implementations, we provide package specification and body templates that may be used as the basis of new development. In the remainder of this section, we describe those templates and discuss the implementation of one of the predefined channels, *FIFO_Channel*.

8.1 The Channel Package Specification Template

```
with Table_Types;
with System; use System;
with Durra_Interface_Types;

package <insert name>_Channel is

  -- FUNCTION
  -- This is the template from which additional channel implementation
  -- specifications should be constructed.

  package TT renames Table_Types;
  package DT renames Durra_Interface_Types;

  --****************************************************************
  --                                TYPES
  --****************************************************************

  subtype Port_Range is DT.Port_ID_Range range 1..<#portsas should all imple
mentations upper bound>;

  task type Channel_Task is

    pragma Priority (Priority'LAST-1);  -- this can be changed.

    entry Initialize (The_Link : in TT.Link_Table_Ptr);

    entry Finish;

    entry Get_Port (
            The_Port             : in     TT.Port_Table_Ptr;
            Data_Location        : in     System.Address;
```

```
            Data_Size            :    out NATURAL;
            Data_Type_ID         :    out DT.Type_ID_Range_Plus_Null;
            Completed            :    out BOOLEAN;
            Blocking             : in     BOOLEAN);

    entry Get_Port_Return(Port_Range)(
            Size_of_Data  :    out NATURAL;
            Type_ID       :    out DT.Type_ID_Range_Plus_Null);

    entry Send_Port (
            The_Port             : in     TT.Port_Table_Ptr;
            Data_Location        : in     System.Address;
            Data_Size            : in     POSITIVE;
            Data_Type_ID         : in     DT.Type_ID_Range_Plus_Null;
            Completed            :    out BOOLEAN;
            Priority             : in     DT.Message_Priority_Range;
            Blocking             : in     BOOLEAN);

    entry Send_Port_Return(Port_Range);

    entry Test_Input_Port (
            The_Port           : in     TT.Port_Table_Ptr;
            Type_of_Next_Input :    out DT.Type_ID_Range_Plus_Null;
            Size_of_Next_Input :    out NATURAL;
            Inputs_Available   :    out NATURAL);

    entry Test_Output_Port (
            The_Port        : in     TT.Port_Table_Ptr;
            Slots_Available :    out NATURAL);


  end Channel_Task;

  type Channel_Ptr is access Channel_Task;

end <insert name>_Channel;
```

Durra channel implementations are required to have almost identical specifications. The only aspects of the specification that an implementor is allowed to change are:

- the name of the package (required change),
- the upper bound of the subtype *Port_Range* (required change), and
- the static priority of the *Channel_Task* type (permitted change).

The upper bound of the subtype *Port_Range* should be set to the larger of the maximum number of input ports or the maximum number of output ports that will be permitted to connect to the channel. This upper bound should be as small as is reasonable, since it determines the cardinality of the families of *Get_Port_Return* and *Send_Port_Return* entries in the *Channel_Task* type.

We consider it wise to leave the priority of the *Channel_Task* type as defined unless there is some compelling reason to change it. In the Durra runtime, the highest Ada priority is reserved for tasks involved in performing reconfigurations. We have assigned the next highest priority to channel implementations. The reason is that a *Channel_Task* is required to have *reactive*

behavior, i.e., the task has no work to do unless it is responding to a request from some other task. Because of this feature, high priority channels will not be in a position to preempt lower priority tasks for potentially unbounded time. Thus, there is no harm in giving them high priority. Conversely, low priority channels are more likely to fall behind requests from higher priority tasks, potentially resulting in more blocking of producers and consumers and therefore higher message delivery overhead.

8.2 The Channel Package Body Template

```
with Storage_Types;
with Storage_Manager;
with Channel_Support;

package body <insert_name>_Channel is

  -- FUNCTION
  -- This is the template for the package body of new channel
  -- implementations for the Durra environment.

--  package TT renames Table_Types in spec
--  package DT renames Durra_Interface_Types in spec
  package ST renames Storage_Types;
  package SM renames Storage_Manager;
  package CS renames Channel_Support;

  use Table_Types;

  task body Channel_Task is
    -- <local declarations of buffer space, etc.>
  begin
    -- Make sure initialization is the first service request filled.
    accept Initialize (The_Link : in TT.Link_Table_Ptr) do
      -- <Initialization actions>
    end Initialize;

    loop
      select
        -- this allows reinitialization
        accept Initialize (The_Link : in TT.Link_Table_Ptr) do
          -- <Initialization actions>
        end Initialize;
      or
        accept Finish;
        exit;
      or
        accept Get_Port (
            The_Port             : in     TT.Port_Table_Ptr;
            Data_Location        : in     System.Address;
            Data_Size            :    out NATURAL;
            Data_Type_ID         :    out DT.Type_ID_Range_Plus_Null;
            Completed            :    out BOOLEAN;
            Blocking             : in     BOOLEAN) do
          -- <forwarding of message to caller or blocking of caller.>
          -- <this section must include a test to see if a producer task>
          -- <has been blocked waiting for a slot in the buffer to open.>
```

```
          -- <if so, then the channel must accept the Send_Port_Return>
          -- <entry call from the producer task so that it will be unblocked.>
        end Get_Port;
      or
        accept Send_Port (
            The_Port            : in     TT.Port_Table_Ptr;
            Data_Location       : in     System.Address;
            Data_Size           : in     POSITIVE;
            Data_Type_ID        : in     DT.Type_ID_Range_Plus_Null;
            Completed           :    out BOOLEAN;
            Priority            : in     DT.Message_Priority_Range;
            Blocking            : in     BOOLEAN) do
          -- <receipt and storage or forwarding of message happens here.>
          -- <this section must include a test to see if a consumer task>
          -- <has been blocked waiting for a message to arrive.>
          -- <if so, then the channel must accept the Get_Port_Return>
          -- <entry call from the consumer task so that it will be unblocked.>
        end Send_Port;
      or
        accept Test_Input_Port (
            The_Port           : in     TT.Port_Table_Ptr;
            Type_of_Next_Input :    out DT.Type_ID_Range_Plus_Null;
            Size_of_Next_Input :    out NATURAL;
            Inputs_Available   :    out NATURAL) do
          --<check for available data, return information>
        end Test_Input_Port;
      or
        accept Test_Output_Port (
            The_Port        : in     TT.Port_Table_Ptr;
            Slots_Available :    out NATURAL) do
          --<check for available storage space, return information>
        end Test_Output_Port;
      end select;
    end loop;
  end Channel_Task;

end <insert_name>_Channel;
```

Unlike their specifications, the package bodies of channel implementations will vary widely according to their desired behavior. They will all be structurally similar, though, for which reason we have provided a package body template.

In each implementation, the *Channel_Task* shall have the following overall structure:

- local data declarations
- an accept statement for the entry Initialize, to ensure that no other entries are accepted until the channel has been initialized
- a loop surrounding a selective wait statement with accept alternatives for all entries (possibly excepting the *Get_Port_Return* and *Send_Port_Return* families of entries)

The following restrictions are placed on channel implementors:

- no package-level variables are allowed
- no executable statements may occur outside of the various rendezvous

Package-level data is prohibited because multiple *Channel_Task* objects may be created from the same channel package. Concurrent access of package-level data could cause state inconsistencies. The restriction of executable statements to within rendezvous guarantees that *Channel_Tasks*, which have been assigned a high priority, will not preempt the processor when not servicing a client.

A new channel implementation is expected to use at least the Durra runtime's storage management and data transfer facilities, provided as packages *Storage_Manager* and *Channel_Support*. The use of these packages will be described next in our discussion of an example channel implementation.

8.3 Example: Our *FIFO_Channel* Implementation

The *FIFO_Channel* is a channel with one input port and one output port. The implementation guarantees to preserve the arrival ordering of messages in the ordering of message outputs.

The following discussion interleaves fragments of the Ada implementation of the *FIFO_Channel* with text describing the important features of the fragment. The conjunction of all the fragments forms a legal Ada package body.

```
with Data_Queues;
with Storage_Types;
with Storage_Manager;
with Channel_Support;
with Table_Manager;

package body FIFO_Channel is

--  package TT renames Table_Types in spec
--  package DT renames Durra_Interface_Types in spec
  package DQ renames Data_Queues;
  package ST renames Storage_Types;
  package SM renames Storage_Manager;
  package CS renames Channel_Support;
  package TM renames Table_Manager;
```

All the packages "withed" in this package are supplied as part of the Durra distribution. The *Storage_Types* and *Storage_Manager* package provide a storage management facility that prevents the unchecked growth of allocated storage in the Durra runtime. The *Data_Queues* package is an instance of an abstract FIFO queue. The elements of the queue are access objects pointing to Durra untyped data block records. The *Table_Manager* package provides query functions for the tables generated by the Durra compiler for specific clusters. The *Channel_Support* package provides operations to abstract the process of transferring data from one thread of control to another (the transfer may be intra-cluster or inter-cluster.)

```
  use Table_Types;

  task body Channel_Task is
    The_Queue : DQ.Queue;
    The_Data  : ST.Queue_Element_Ptr;
```

```
The_Type  : TT.Type_Table_Ptr;
Producer_Blocked : BOOLEAN := FALSE;
Consumer_Blocked : BOOLEAN := FALSE;
Blocked_Process : TT.Process_Table_Ptr;
New_Q_Element : ST.Queue_Element_Ptr;
```

We use a queue to implement the message buffer for this channel. Flags are included to keep track of whether the producer or consumer processes are blocked. If one of them is blocked (only one may be blocked at any given time), then *Blocked_Process* identifies it.

```
begin
  -- Make sure initialization is the first service request filled.
  accept Initialize (The_Link : in TT.Link_Table_Ptr) do
    DQ.Clear(The_Queue);
    DQ.Set_Bound(The_Queue, The_Link.Buffer_Size);
  end Initialize;
```

The *Initialize* entry must be accepted before any other entries. It causes the queue to be cleared and the queue bound to be set to maximum buffer size for the link. The default buffer size is one message, but this can be overridden with the *bound* attribute in the Durra description of the link. Note that the size of a buffer for any link is expressed in terms of the number of messages that the link can store. This is independent of the size of the messages themselves.

```
loop
  select
    -- this allows reinitialization
    accept Initialize (The_Link : in TT.Link_Table_Ptr) do
      DQ.Clear(The_Queue);
      DQ.Set_Bound(The_Queue, The_Link.Buffer_Size);
    end Initialize;
  or
    accept Finish;
    exit;
```

In the main loop of the *Channel_Task*, which is executed until the *Finish* entry is accepted, we allow acceptance of *Initialize* again in order to provide for reinitialization of the link.

```
  or
    accept Get_Port (
        The_Port             : in     TT.Port_Table_Ptr;
        Data_Location        : in     System.Address;
        Data_Size            :    out NATURAL;
        Data_Type_ID         :    out DT.Type_ID_Range_Plus_Null;
        Completed            :    out BOOLEAN;
        Blocking             : in     BOOLEAN) do
      if DQ.Is_Empty(The_Queue) then
        if Blocking then
          Consumer_Blocked := TRUE;
          Blocked_Process := The_Port.Owner_Process;
          Blocked_Process.Blocked_Data_Buffer := Data_Location;
        end if;
        Data_Type_ID := DT.NULL_TYPE_ID;
        Data_Size := 0;
        Completed := FALSE;
```

```
      else
        CS.Transfer_Data_From_Queue(
             The_Port.Owner_Process,
             Data_Location,
             Data_Size,
             Data_Type_ID,
             The_Queue);
        Completed := TRUE;
        if Producer_Blocked then
          Producer_Blocked := FALSE;
          accept Send_Port_Return(1);
        end if;
      end if;
    end Get_Port;
```

A consumer process indirectly calls the *Get_Port* entry of a link when it wishes to receive a message. If the message queue is empty, then the link returns values indicating that the request could not be fulfilled. If the request is a blocking version, then the link keeps track of the process that is blocked and the location where it wants to receive the message. If there is at least one message in the queue, then the message at the front of the queue is transferred to the calling process. Since the removal of a message from the queue frees space in the link buffer, the link must check to see whether the producer attached to its input port is blocked waiting to deliver a message. If so, the link accepts the *Send_Port_Return* entry and unblocks the producer.

```
  or
    accept Send_Port (
        The_Port             : in     TT.Port_Table_Ptr;
        Data_Location        : in     System.Address;
        Data_Size            : in     POSITIVE;
        Data_Type_ID         : in     DT.Type_ID_Range_Plus_Null;
        Completed            :    out BOOLEAN;
        Priority             : in     DT.Message_Priority_Range;
        Blocking             : in     BOOLEAN) do
      if Consumer_Blocked then
        Consumer_Blocked := FALSE;
        CS.Transfer_Data_from_Process_to_Process(
             The_Port.Owner_Process,
             Blocked_Process,
             1,
             Data_Location,
             Data_Size,
             Data_Type_ID);
        accept Get_Port_Return(1)(
                 Size_of_Data : out NATURAL;
                 Type_ID      : out DT.Type_ID_Range_Plus_Null) do
          Size_of_Data := Data_Size;
          Type_ID := Data_Type_ID;
        end Get_Port_Return;
        Completed := TRUE;
      else
        if not Blocking then
          --Check to make sure operation will complete without blocking
          --before actually accepting the data.
          if DQ.Next_Item_Would_Fill(The_Queue) then
```

```
            Completed := FALSE;
            return;
          end if;
        end if;
        New_Q_Element := CS.Transfer_Data_from_Process(
                             The_Port.Owner_Process,
                             Data_Location,
                             Data_Size,
                             Data_Type_ID,
                             1);
        DQ.Add (New_Q_Element, The_Queue);
        if DQ.Is_Full(The_Queue) then
          -- Will not be TRUE for non-blocking Send_Ports.
          Blocked_Process := The_Port.Owner_Process;
          Producer_Blocked := TRUE;
          Completed := FALSE;
        else
          Completed := TRUE;
        end if;
      end if;
    end Send_Port;
```

A producer process indirectly calls the *Send_Port* entry of a link when it wishes to send a message. If the consumer process is already blocked waiting for a message, the message is transferred directly to the consumer's address space. The consumer is unblocked by accepting its *Get_Port_Return* entry. If the consumer isn't waiting for a message already, then the behavior depends on whether the producer has requested a blocking or non-blocking *Send_Port*. If the request is a non-blocking version, then the link checks to see if this message would fill the queue and cause the producer to block. If it would, then the link returns a value indicating that the service request could not be completed. Otherwise, the message is transferred from the producer address space to the link address space and added to the end of the queue. If the queue is then full, the producer process will be blocked.

```
  or
    accept Test_Input_Port (
        The_Port           : in     TT.Port_Table_Ptr;
        Type_of_Next_Input :    out DT.Type_ID_Range_Plus_Null;
        Size_of_Next_Input :    out NATURAL;
        Inputs_Available   :    out NATURAL) do
      Inputs_Available := DQ.Length_Of(The_Queue);
      if not DQ.Is_Empty(The_Queue) then
        The_Data := DQ.Front_Of(The_Queue);
        Type_of_Next_Input := The_Data.Data_Handle.Data_Type_ID;
        Size_of_Next_Input := The_Data.Data_Handle.Data_Size;
      else
        Type_of_Next_Input := DT.NULL_TYPE_ID;
        Size_of_Next_Input := 0;
      end if;
    end Test_Input_Port;
```

A consumer process indirectly calls the *Test_Input_Port* entry of a link when it wishes to test for the availability of a message or the size and type of the next message from the port. If the link buffer is empty, then the link returns values indicating that no message is available. Oth-

erwise, the link returns the number of messages available and the Durra type and size of the first message in the queue.

```
      or
        accept Test_Output_Port (
            The_Port        : in     TT.Port_Table_Ptr;
            Slots_Available :    out NATURAL) do
          Slots_Available := DQ.Bound(The_Queue) - DQ.Length_Of(The_Queue);
        end Test_Output_Port;
      end select;
    end loop;
  end Channel_Task;

end FIFO_Channel;
```

A producer process indirectly calls the *Test_Output_Port* entry when it wishes to know how many messages it may send to the specified port before it might block. The link returns the number of empty message spaces in its buffer.

See the library $DURRA_ROOT/channels for more example specifications and implementations of Durra channels.

9 Durra Configuration Files

Durra applications use a *configuration file* to specify the mapping of clusters to physical processors. A configuration file may be specified as an argument to the Durra compiler, in which case the configuration information is included in the tables generated in the body of package Tables. If no compile-time configuration is specified, then the file must be supplied at runtime as an argument to the cluster programs. A user is permitted to supply a runtime configuration file argument even if a configuration was specified to the compiler. In that case, the runtime configuration specification overrides the one compiled.

A configuration fie is a text file containing instructions to the Durra runtime. There are two types of instructions in a configuration file, the **processor** instruction and the **cluster** instruction. All processor instructions must precede any cluster instructions in the file.

The processor instruction has the following form:

> **processor** *processor-name* { *processor-group-name-list* }

The *processor-name* must be the name of a physical processor in Internet format (e.g., ftp.sei.cmu.edu). The processor name may be followed by a list of logical names for the processor. The logical names are then considered equivalent to the actual name, unless the logical name applies to more than one actual processor, in which case the name is equivalent to the name of any one of the group of processors to which it applies. Here are examples of processor instructions:

> **processor** ftp.sei.cmu.edu sun4 mysun
> **processor** moby.sei.cmu.edu sun4 yoursun

Given the above processor instructions, a cluster assigned to logical processor mysun could only be assigned to physical processor ftp.sei.cmu.edu. A cluster assigned to logical processor sun4, however, could be assigned to either physical processor. In such a case Durra assigns the cluster to the host with the fewest clusters already assigned to it.

The cluster instruction has the following form:

> **cluster** *cluster-name cluster-host-name* { *cluster-start-up-command* }

The *cluster-name* must be the name of a cluster defined in the application description to which the configuration is being applied. The *cluster-host-name* must be either the actual name or a logical name of a physical processor identified in a previous processor instruction. Depending on when the configuration file is specified, either the Durra compiler or the Durra runtime apply consistency checking to configuration specifications. For example, if Durra processes A and B are assigned to logical processors X and Y in their respective descriptions, and A and B are assigned to cluster C in the application description, then the configuration file for the application must contain at least one processor instruction identifying a physical processor to which both logical names X and Y applies. The file must also contain a cluster instruction assigning

cluster C to that processor. Otherwise, the Durra compilation/execution terminates with an error on the assumption that the user is inadvertently attempting to execute a process on a processor for which it was not intended.

The *cluster-start-up-command* is the command that should be used by the **Durra_Launcher** (see Section 4.8) on the specified host to execute the cluster. The command is optional; it never needs to be supplied for the master cluster, since that cluster will always be started by hand from a shell. It may also be omitted from the cluster instructions for all clusters if the clusters are to be started independently, without use of the launcher; see Section 11. The command must be supplied for all non-master clusters if the launcher is to be used.

If the cluster requires an independent terminal display then the command must cause the execution of an **xterm(1)** process that indirectly executes the cluster. The full path names of all files to be executed or read must be specified, since the Durra runtime will not search the user's path for files. Following are some examples of cluster instructions that might be used with our *dynamic_producer_consumer* example:

cluster cl1 sun4 /usr/Durra/Manual-User/example/dpc/cl1/.sun4/cluster

cluster cl1 sun4 /usr/local/X11R4/bin/xterm -title cl1 -g =80x20-1+0 -e /usr/projects/Durra/Manual-User/example/dpc/cl1/.sun4/cluster -c/usr/projects/hetsim/Durra/Manual-User/example/dpc.config

cluster cl2 sun4 /usr/local/X11R4/bin/xterm -title cl2 -e a.db -L /usr/projects/Durra/Manual-User/example/dpc/cl2/.sun4 /usr/projects/Durra/Manual-User/example/dpc/cl2/.sun4/cluster

The first example simply executes the cluster. The second example executes an **xterm(1)**, giving it the cluster file name as the name of the program to execute within the window. The argument to the cluster program is the name of the configuration file that the cluster should read at runtime. The third example executes an xterm(1) and tells it to execute the Verdix Ada debugger on the cluster program. The configuration file name must be specified within the debugger if it was not already compiled in.

The current limit on the length of lines in the configuration file is 500 characters.

10 Compiling and Linking a Durra Application

Once component implementations and specifications have been written, a Durra application can be compiled and linked into a set of executable programs, one for each cluster identified in the application. We will continue to use the *dynamic_producer_consumer* example to illustrate this process.

10.1 Durra Compilation and Makefiles

When a library of Durra components is compiled for the first time, it is necessary to create a Durra library and compile the components by hand in some partial ordering that respects component dependencies. In the *dynamic_producer_consumer*, for example, the declaration of the type *message* depends upon the types *scalar* and *byte*, so those two type declarations must be compiled before the message type declaration. We first create a Durra library to contain the component definitions:

dlibrary -c

This command creates the file .DLIBRARY in the current working directory. Since we also intend to use the predefined channel description FIFO, we need to add the predefined channel library to our Durra library path:

dlibrary -a $DURRA_ROOT/channels/durra

We can now compile the Durra descriptions in any legal partial ordering:

dall byte.durra
dall scalar.durra
dall message.durra
dall producer.durra
dall consumer.durra
dall -g -rdpc **-c**dpc.config dyn_prod_cons.durra

For each successful compilation, the following are created:

- an intermediate representation of the component, stored in the file with the same root name and the extension ".TREE"
- an entry in the .DLIBRARY file pointing to the ".TREE" file
- a version control file with the same root name and the extension ".MAKE"

Here is the file "producer.durra.MAKE", which is created when "producer.durra" is compiled:

```
CLROOT =
DFLAGS =

all:        durra

durra:        dependencies producer.durra.TREE
```

```
dependencies:
       cd /u/fs4a/c/hetsim/durra/examples/dyn_prod_cons; dmake message.durra

producer.durra.TREE: producer.durra \
          /u/fs4a/c/hetsim/durra/examples/dyn_prod_cons/message.durra.TREE
        $(DALL_VERSION) $(DFLAGS) producer.durra

cleanentry:
        rm producer.durra.TREE
```

Each ".MAKE" file generated by the Durra compiler will have at least the macro definitions and target entries shown above. Here is a brief description of each:

- CLROOT macro : when defined, defines the cluster root directory
- DFLAGS macro : defines the list of optional arguments to be applied when compiling the Durra component
- "all" target : points only to the "durra" target for non-application descriptions
- "durra" target : points to the "dependencies" target and the target for the component's intermediate representation
- "dependencies" target: causes **dmake** to be called for all descriptions on which this component directly depends. Indirect dependencies are handled by the ".MAKE" files of the direct dependencies, so that a closure of dependencies is formed
- the "*.TREE" target: checks for any change in the component source file or the ".TREE" files of any direct dependencies. If any change is found, then recompiles the component
- "cleanentry" target : removes the ".TREE" file for the component

Components for which cluster code is generated have more complex ".MAKE" files. Here is the file "dyn_prod_cons.durra.MAKE":

```
CLROOT = dpc
DFLAGS = -g -r$(CLROOT) -cdpc.config
ARCH = .‘arch‘
OPT = -v -All

all:        durra clusters

durra:      dependencies dyn_prod_cons.durra.TREE

dependencies:
     cd /u/fs4a/c/hetsim/durra/channels/durra; dmake fifo.durra
     cd /u/fs4a/c/hetsim/durra/examples/dyn_prod_cons; dmake message.durra
     cd /u/fs4a/c/hetsim/durra/examples/dyn_prod_cons; dmake consumer.durra
     cd /u/fs4a/c/hetsim/durra/examples/dyn_prod_cons; dmake producer.durra

dyn_prod_cons.durra.TREE: dyn_prod_cons.durra \
      /u/fs4a/c/hetsim/durra/channels/durra/fifo.durra.TREE \
      /u/fs4a/c/hetsim/durra/examples/dyn_prod_cons/message.durra.TREE \
      /u/fs4a/c/hetsim/durra/examples/dyn_prod_cons/consumer.durra.TREE \
      /u/fs4a/c/hetsim/durra/examples/dyn_prod_cons/producer.durra.TREE \
           dpc.config
        $(DALL_VERSION) $(DFLAGS) dyn_prod_cons.durra
```

```
clusters:
        cd $(CLROOT)/cl1; make OPT=”$(OPT)”
        cd $(CLROOT)/cl2; make OPT=”$(OPT)”

cleanentry:
        rm dyn_prod_cons.durra.TREE

clusterworld:
        cd $(CLROOT)/cl1; make world OPT=”$(OPT)”
        cd $(CLROOT)/cl2; make world OPT=”$(OPT)”

world:        cleanentry durra clusterworld
```

Here is a brief description of the additional entries found in these more complicated “.MAKE” files:

- ARCH macro : the architecture for which we are compiling the Ada source
- OPT macro : the list of options to use when doing Ada compilations (this is VADS-specific)
- “all” target : points to the “clusters” target as well as the “durra” target
- “clusters” target : causes each of the cluster executables to be checked to see if recompilation is required
- “clusterworld” target : forces each of the cluster executables to be recompiled in a fresh Ada library
- “world” target : forces a recompilation of both the application description and the cluster executables

Once the “.MAKE” files have been created, it is no longer necessary or desirable to use the **dall** command directly. The **dmake** command (see Section 4.5) should be used instead.

10.2 Cluster Compilation

As a result of the commands performed in the example in the previous section, the subdirectory “dpc” will have been created. That directory will in turn have subdirectories “cl1” and “cl2”, corresponding to the names of the clusters as specified in the application description. Each of those subdirectories contains a file named “TablesB.a”, which in each case contains a Durra compiler-generated, cluster-specific version of the *Tables* package body. In order to facilitate rapid recompilation of the Ada portion of a Durra application, we need to supply a “Makefile” for each of these cluster subdirectories. Since the cluster “Makefile” is almost identical in every case, we supply a template (in the file “$DURRA_ROOT/misc/Cluster_Makefile”) that needs to be modified in only one place. Where indicated, the user must add the names of any Ada libraries from which component implementations are to be imported. After making this change, the user saves the new version of the Makefile and distributes it to all cluster subdirectories via the **dmklib** command (see Section 4.4). For example, assuming the current working direc-

tory is directory “dpc” and the new version of “Cluster_Makefile” was saved in the file “my_makefile”, then we would use the command:

dmklib -m my_makefile

We are now ready to compile and link the Ada code. Returning to the Durra source directory, we issue the command:

dmake dyn_prod_cons world

This command checks to make sure that the Durra components are up to date, creates the Ada libraries for the cluster programs, compiles the Tables package bodies and any dependent Ada units that are not up to date, and links each of the cluster programs. We are now ready to execute the application.

11 Executing a Durra Application

Executing a Durra application is easy. There are two steps in the process. The first is to make sure that the **Durra_Launcher** (see Section 4.8) is running on each processor required for the application execution. The second is to start the master cluster (and optionally, the other clusters).

Since each executable Durra cluster program is placed in a file named “cluster”, each cluster must be in a separate directory. The user should change the current working directory to be the directory containing the master cluster (which, as of this version of Durra, is always the first cluster named in the application description). The user should then type the **cluster** command:

cluster { **-n** } { **-c***configuration-file-name* } { **-d***debug-level-number* }

If the **cluster** command is entered with no optional arguments, then the master cluster will be started. It will terminate with an error if no configuration file (see Section 9) was specified for it at compile time. Otherwise, it will attempt to make contact with the launchers on all machines identified in the configuration file. It will instruct the launchers to start the subordinate clusters on the machines to which the clusters are assigned.

If the **-n** (meaning “no launcher”) flag is specified when the master cluster is invoked, then the master cluster will not attempt to automatically start the subordinate clusters. The subordinate clusters must then be started by hand from other shells. When starting clusters in this fashion, it is best to start them in the order in which they are specified in the application description. The reason is that, in the UNIX implementation, the higher numbered clusters attempt to connect to the lower numbered clusters, which are expected to be waiting for the connection. If the higher numbered cluster can’t make the connection, it will keep attempting to do so for some time-out period (currently 10 seconds), issuing an error message each time the connection attempt fails. So, if a user were to start the higher numbered cluster first, then a connection could be made only if user also gets the lower-numbered cluster started within the time-out period. In the meantime, the display will fill with error messages. The **-n** flag has no effect when specified for any cluster other than the master cluster.

If the **-c** flag is specified when a cluster is invoked, the cluster will use the specified file as the configuration for the purposes of this execution. A runtime-specified configuration overrides any compile-time configuration specification. If the clusters are started independently, then each cluster must be provided with the same configuration file from the command line. Otherwise, the same file name must be specified for each subordinate cluster in the configuration file itself, as part of the “command” field of the **cluster** instruction. Note that Durra assumes that each cluster will be able to read a common configuration file, even if the clusters are on separate machines. The implication of this assumption is that either all processors have access to a common networked file system or that a copy of the configuration file is accessible on each processor’s file system. If neither of these conditions is satisfiable, then the configuration must be specified at compile-time.

If the **-d** (meaning “debug”) flag is specified then the argument associated with the flag must be one of the numbers 0, 1, and 2, which correspond to “no debugging”, “intermediate debugging”, and “full debugging”, respectively. The default behavior is “no debugging.” When either of the other two levels is specified, then the cluster will log trace information to a file in the directory specified by the environment variable DURRA_LOG_DIR (see Section 3.1). “Intermediate debugging” gives trace information at the level of *Durra_Interface* operations (e.g., “Send_Port completed on port p.output”). “Full debugging” gives trace information at the level of underlying network operations (e.g., “Received: mailbox 2, data of length 16”). Log files for “full debugging” will be quite large. Note that the debugging we are talking about here is debugging of the Durra runtime, via traces included in the Durra runtime. Debugging of application code must be done using your Ada debugger.

Appendix A VADS Dependencies in Durra

Durra was developed using the Verdix Ada Development System. Although the vast majority of the Durra source code is portable between Ada compilers, there are a small number of dependencies on non-standard facilities provided with VADS.

It is well-known that the methods for interfacing to other languages (via *pragma Interface*) vary slightly according to compiler vendor. The UNIX implementation of the Durra runtime includes a large number of interfaces with UNIX system services written in C. These system call interfaces may require modification for other compilers.

The implementations of various parts of Durra also depend on some non-standard packages provided with VADS. The following is a list of these packages, and the Durra units that depend on them. It is likely that other implementations will supply non-standard packages with equivalent functionality.

- *c_strings* : provides an abstraction of the C (* char) type. Used by the package bodies *Operating_System_Interface* (in “runtime/unix” directory) and *Utilities* (in “compiler” directory), as well as the Durra compiler main unit (in “compiler” directory).
- *Current_Exception* : provides a way of identifying by the name an exception caught in an “others” handler. Used by package body *Process_Shell* (in “runtime/lib” directory).
- *Math* : provides a library of mathematical functions. Used by package *Random* (in “adalib” directory).
- *U_Env* : provides *argv/argc* to an Ada program. Used by the package bodies *Operating_System_Interface* (in “runtime/unix” directory) and *Utilities* (in “compiler” directory), as well as the Durra compiler main unit (in “compiler” directory).
- *UNIX* : provides an abstraction of some UNIX system services. Use by the compiler main unit (in “compiler” directory).
- *Unsigned_Types* : provides unsigned numeric types. Used by the package body *Operating_System_Interface* (in “runtime/unix” directory”) and the main unit of the **Durra_Launcher** (in “launcher” directory).
- *v_i_bits* : provides an abstraction for bit-level logical operations. Used by the package body *Operating_System_Interface* (in “runtime/unix” directory”).
- *v_i_sema* : provides a semaphore abstraction. Used by package body *Network* (in “runtime/lib” directory).
- *V_Semaphores* : provides a different semaphore abstraction. Used by package bodies *Storage_Manager* and *Tasking_IO* (both in “runtime/lib” directory).

The “makefiles” supplied with the Durra libraries and the “.MAKE” files generated by the Durra compiler include explicit VADS software tools commands. The makefiles we supply would have to be changed and the compiler modified to generate different “.MAKE” files in order to support a different Ada compilation system.

Appendix B The *Durra_Interface* Package Specification

Following is the complete specification of the package Durra_Interface, the API to Durra runtime services:

```
------------------------------------------------------------------------
--|                        Software Engineering Institute
--|                    Durra Language and Runtime Environment
--|
--| The Durra Language and Runtime Environment are distributed under the
--| terms of a Memorandum of Understanding or a Licensing Agreement. Use or
--| distribution of the software is governed by the terms of the
--| appropriate Memorandum of Understanding or Licensing Agreement.
--|
--|        (c) Carnegie Mellon University 1989, all rights reserved
------------------------------------------------------------------------

with System; use System;
with Durra_Interface_Types;

package Durra_Interface is
  ----------------------------------------------------------------------
  --| Durra Runtime Interface
  --|
  --| This package provides an interface to Durra services for Durra
  --| application processes.

  package DT renames Durra_Interface_Types;

  --
**************************************************************************
  --                                  TYPES
  --
**************************************************************************

  type Type_ID        is private;
  type Input_Port_ID  is private;
  type Output_Port_ID is private;
  type Process_ID     is private;

  subtype Signal_Range is DT.Signal_Range;
  subtype Message_Priority is DT.Message_Priority_Range;

  --
**************************************************************************
  --                                CONSTANTS
  --
**************************************************************************

  NULL_TYPE_ID          : constant Type_ID;
  NULL_MESSAGE_PRIORITY : constant Message_Priority
                                    := DT.NULL_MESSAGE_PRIORITY;
  --
**************************************************************************
  --                               EXCEPTIONS
```

```
  --
*************************************************************************

  Bad_Port_Name : exception;
  -- Raised by Get_PortId when Port_Name is not defined for this Process.
  Bad_Port_ID   : exception;
  -- Raised when the Port_ID is undefined for the given Process_ID or the
  -- port is the wrong direction for the operation.
  Bad_Process_ID   : exception;
  -- Raised when the Process_ID is undefined in the application.
  Bad_Type_Name : exception;
  -- Raised by Get_TypeId when Type_Name is not defined for the
  -- application.
  Bad_Type_ID   : exception;
  -- Raised when the Type_ID is undefined in the application or when the
  -- Type_ID is not appropriate for a specified port.
  Uninitialized : exception;
  -- Raised by any of the interface services (other than Init) when called
  -- before the process has done a call to Init.
  Already_Initialized: exception;
  -- Raised by Init when it is called more than once.

  --
*************************************************************************
  --                          VISIBLE SUBPROGRAMS
  --
*************************************************************************

  --**************
  -- TIME ROUTINES
  --**************

  function Get_Application_Time return DURATION;
    -- PARAMETERS
    --   None
    -- DESCRIPTION
    --   Returns the time elapsed since the start of the application of
    --   which this process is a component.
    -- EXCEPTIONS RAISED
    --   None

  --
*************************************************************************

  function Get_Process_Time (Process : in Process_ID) return DURATION;
    -- PARAMETERS
    --   Process           : the process identifier
    -- DESCRIPTION
    --   Returns the time elapsed since the start of this process.
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID

  --
*************************************************************************

  function Get_Day_Time return DURATION;
    -- PARAMETERS
    --   None
```

```
    -- DESCRIPTION
    --   Returns the time elapsed since midnight of the current day.
    -- EXCEPTIONS RAISED
    --   Uninitialized

  --****************************
  -- INTERFACE TO DURRA EXECUTIVE
  --****************************

  procedure Finish (Process : in Process_ID);
    -- PARAMETERS
    --   Process           : the process identifier
    -- DESCRIPTION
    --   Tell the Durra runtime that this Process is finished.  No other
    --   interface calls are permitted once this call has been issued.
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID

  --
*************************************************************************

  function Get_Attribute (Process        : in Process_ID;
                          Attribute_Name : in STRING) return STRING;
    -- PARAMETERS
    --   Process           : the process identifier
    --   Attribute_Name    : the (case-insensitive) name of an attribute
    --                       for which the value should be retrieved.
    -- DESCRIPTION
    --   Returns the STRING value of the process attribute specified by
    --   Attribute_Name.  If no such attribute exists, returns the NULL
    --   string.
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID

  --
*************************************************************************

  procedure Get_Port (Process       : in     Process_ID;
                      Port          : in     Input_Port_ID;
                      Data          : in     System.Address;
                      Data_Size     :    out NATURAL;
                      Data_Type     :    out Type_ID);
    -- PARAMETERS
    --   Process           : the process identifier
    --   Port              : the identifier of the input port on which the
    --                       Get is to be performed
    --   Data              : the address of a buffer where the incoming
    --                       data will be placed
    --   Data_Size         : the size (in bytes) of the received data
    --   Data_Type         : the Durra type identifier of the data
    --                       received.
    -- DESCRIPTION
    --   Get a message at port Port and deposit it at location Data.
    --   It is up to the application task to make sure that the buffer at
    --   location Data is large enough to hold the incoming message.  The
    --   call blocks if there is currently no data to be retrieved at
    --   Port.
```

```
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID
    --   Bad_Port_ID

  --
*************************************************************************

  procedure Get_Port (Process   : in     Process_ID;
                      Port      : in     Input_Port_ID;
                      Data      : in     System.Address;
                      Data_Size :    out NATURAL;
                      Data_Type :    out Type_ID;
                      Got_Data  :    out BOOLEAN);
    -- PARAMETERS
    --   Process           : the process identifier
    --   Port              : the identifier of the input port on which the
    --                       Get is to be performed
    --   Data              : the address of a buffer where the incoming
    --                       data will be placed
    --   Data_Size         : the size (in bytes) of the received data
    --   Data_Type         : the Durra type identifier of the data
    --                       received
    --   Got_Data          : flag indicating whether or not any data was
    --                       actually received.
    -- DESCRIPTION
    --   Non-blocking version of Get_Port; if there is a message available,
    --   get it as in the blocking version and return Got_Data = TRUE;
    --   otherwise, return Got_Data = FALSE.
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID
    --   Bad_Port_ID

  --
*************************************************************************

  procedure Get_PortId (Process     : in     Process_ID;
                        Port_Name   : in     STRING;
                        Port        :    out Input_Port_ID;
                        Data_Size   :    out NATURAL;
                        Port_Type_ID:    out Type_ID);
    -- PARAMETERS
    --   Process           : the process identifier
    --   Port_Name         : the name of the port whose ID is wanted
    --   Port              : the identifier of the input port specified by
    --                       Port_Name
    --  Data_Size          : the max size (in bytes) of the messages that can
    --                       pass through this port.  If the data type for
    --                       this port is an unbounded variable length
    --                       type, then Data_Size is 0 by convention.  If
    --                       the data type is a Durra Union type, then
    --                       Data_Size will be set to the size of the
    --                       largest type in the Union, or to 0 if at least
    --                       one of the types is unbounded variable length.
    --   Port_Type_ID      : the ID of the type associated with the port.
    -- DESCRIPTION
    -- Get the internal Durra ID of the named port, along with the
    -- associated Type_ID and max data size, which are described above.
```

```
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID
    --   Bad_Port_Name

  --
*************************************************************************

  procedure Get_PortId (Process     : in     Process_ID;
                        Port_Name   : in     STRING;
                        Port        :    out Output_Port_ID;
                        Data_Size   :    out NATURAL;
                        Port_Type_ID:    out Type_ID);
    -- PARAMETERS
    --   Process           : the process identifier
    --   Port_Name         : the name of the port whose ID is wanted
    --   Port              : the identifier of the output port specified by
    --                       Port_Name
    --   Data_Size         : the max size (in bytes) of the messages that can
    --                       pass through this port.  If the data type for
    --                       this port is an unbounded variable length
    --                       type, then Data_Size is 0 by convention.  If
    --                       the data type is a Durra Union type, then
    --                       Data_Size will be set to the size of the
    --                       largest type in the Union, or to 0 if at least
    --                       one of the types is unbounded variable length.
    --   Port_Type_ID      : the ID of the type associated with the port.
    -- DESCRIPTION
    -- Get the internal Durra ID of the named port, along with the
    -- associated Type_ID and max data size, which are described above.
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID
    --   Bad_Port_Name

  --
*************************************************************************

  procedure Get_TypeId (Type_Name   : in     STRING;
                        Data_Type   :    out Type_ID;
                        Type_Size   :    out NATURAL);
    -- PARAMETERS
    --   Process           : the process identifier
    --   Type_Name         : the name of the type whose ID is wanted
    --   Data_Type         : the identifier of the type specified by
    --                       Type_Name
    --   Type_Size         : the max size (in bytes) of messages of this
    --                       type.  If the  type is an unbounded variable
    --                       length type, then Type_Size is 0 by
    --                       convention.  If the data type is a Durra Union
    --                       type, then Type_Size will be set to the size
    --                       of the largest type in the Union, or to 0 if
    --                       at least one of the types is unbounded
    --                       variable length.
    -- DESCRIPTION
    -- Get the internal Durra ID of the named type, along with the
    -- type size, which is described above.
    -- EXCEPTIONS RAISED
    --   Uninitialized
```

```
    --   Bad_Type_Name

  --
*************************************************************************

  procedure Init (Task_Sequence_Number : in     POSITIVE;
                  Process_Identifier   :    out Process_ID);
    -- PARAMETERS
    --   Task_Sequence_Number  : the application-unique positive value
    --                           passed to a process by the Durra runtime
    --   Process_Identifier    : the private encoding of
    --                           Task_Sequence_Number, to be used when
    --                           calling all other Durra services.
    -- DESCRIPTION
    -- Establishes presence of this process with the Durra runtime. Must be
    -- the first Durra interface call in any application task.
    -- EXCEPTIONS RAISED
    --   Already_Initialized

  --
*************************************************************************

  procedure Raise_Signal (Process       : in Process_ID;
                          Signal_Number : in Signal_Range);
    -- PARAMETERS
    --   Process           : the process identifier
    --   Signal_Number     : the number of the signal to be raised.  The
    --                       action(s) associated with a signal must be
    --                       specified in the Durra application
    --                       description.
    -- DESCRIPTION
    --   Send a condition signal to the Durra runtime.
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID

  --
*************************************************************************

  procedure Safe (Process : in Process_ID);
    -- PARAMETERS
    --   Process            : the process identifier
    -- DESCRIPTION
    -- Tell the Durra runtime that this Process is safe to reconfigure.
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID

  --
*************************************************************************

  procedure Send_Port (Process   : in Process_ID;
                       Port      : in Output_Port_ID;
                       Data      : in System.Address;
                       Data_Size : in NATURAL;
                       Data_Type : in Type_ID := NULL_TYPE_ID;
                       Priority  : in Message_Priority
                                      := NULL_MESSAGE_PRIORITY);
    -- PARAMETERS
```

```
    --   Process           : the process identifier
    --   Port              : the identifier of the output port on which the
    --                       Send is to be performed
    --   Data              : the address of a buffer where the outgoing
    --                       data is located
    --   Data_Size         : the size (in bytes) of the data
    --   Data_Type         : the Durra type identifier of the data.  The
    --                       default is NULL_TYPE_ID, which by convention
    --                       allows avoidance of port type checking.  This
    --                       feature allows for a more generic application
    --                       component, but some safety and speed is
    --                       sacrificed.
    --   Priority          : the priority of this message.  Priority will
    --                       be ignored unless the port is connected to a
    --                       link which recognizes priorities.  For remote
    --                       communications, Priority will have no effect
    --                       unless the underlying communication protocol
    --                       supports message priorities.
    -- DESCRIPTION
    --   Send data located at address Data to port Port.  The
    --   call blocks if the buffer associated with the port is full.
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID
    --   Bad_Port_ID
    --   Bad_Type_ID

  --
**********************************************************************

  procedure Send_Port (Process   : in     Process_ID;
                       Port      : in     Output_Port_ID;
                       Data      : in     System.Address;
                       Data_Size : in     NATURAL;
                       Data_Sent :    out BOOLEAN;
                       Data_Type : in     Type_ID := NULL_TYPE_ID;
                       Priority  : in     Message_Priority
                                             := NULL_MESSAGE_PRIORITY);
    -- PARAMETERS
    --   Process           : the process identifier
    --   Port              : the identifier of the output port on which the
    --                       Send is to be performed
    --   Data              : the address of a buffer where the outgoing
    --                       data is located
    --   Data_Size         : the size (in bytes) of the data
    --   Data_Sent         : flag indicating whether or not the data was
    --                       actually sent.
    --   Data_Type         : the Durra type identifier of the data.  The
    --                       default is NULL_TYPE_ID, which by convention
    --                       allows avoidance of port type checking.  This
    --                       feature allows for a more generic application
    --                       component, but some safety and speed is
    --                       sacrificed.
    --   Priority          : the priority of this message.  Priority will
    --                       be ignored unless the port is connected to a
    --                       link which recognizes priorities.  For remote
    --                       communications, Priority will have no effect
    --                       unless the underlying communication protocol
    --                       supports message priorities.
```

```
    -- DESCRIPTION
    --   Non-blocking version of Send_Port.  If the queue associated with
    --   the port is full, then no data is sent and Data_Sent is set to
    --   FALSE; otherwise the Send proceeds as in the blocking version and
    --   Data_Sent is set to TRUE.
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID
    --   Bad_Port_ID
    --   Bad_Type_ID

  --
*************************************************************************

  procedure Test_Input_Port
              (Process            : in     Process_ID;
               Port               : in     Input_Port_ID;
               Type_of_Next_Input :    out Type_ID;
               Size_of_Next_Input :    out NATURAL;
               Inputs_Available   :    out NATURAL);
    -- PARAMETERS
    --   Process           : the process identifier
    --   Port              : the identifier of the input port on which the
    --                       Test is to be performed
    --   Type_of_Next_Input: the Durra type id of the next message in the
    --                       queue associated with this port (or
    --                       NULL_TYPE_ID, if Inputs_Available is zero)
    --   Size_of_Next_Input: the size (in bytes) of the next message in the
    --                       buffer associated with this port (or zero, if
    --                       Inputs_Available is zero).
    --   Inputs_Available  : the number of message currently available at
    --                       this port.
    -- DESCRIPTION
    --   Test for the availability of data at the port Port.  If at
    --   least one input is available, return the number of inputs
    --   available and the Durra type and size of the next input;
    --   otherwise, return zero or NULL in all three parameters.
    -- EXCEPTIONS RAISED
    --   Uninitialized
    --   Bad_Process_ID
    --   Bad_Port_ID

  --
*************************************************************************

  procedure Test_Output_Port (Process          : in     Process_ID;
                              Port             : in     Output_Port_ID;
                              Spaces_Available :    out NATURAL);
    -- PARAMETERS
    --   Process           : the process identifier
    --   Port              : the identifier of the output port on which the
    --                       Test is to be performed
    --   Spaces_Available  : the number of empty spaces currently in the
    --                       buffer associated with this port.
    -- DESCRIPTION
    --   Return the number of empty spaces in the buffer associated with
    --   Port.
    -- EXCEPTIONS RAISED
    --   Uninitialized
```

```
    --   Bad_Process_ID
    --   Bad_Port_ID

  --
*************************************************************************

private
  type Type_ID        is new DT.Type_ID_Range_Plus_Null;
  type Input_Port_ID  is new DT.Port_ID_Range;
  type Output_Port_ID is new DT.Port_ID_Range;
  type Process_ID     is new DT.Process_ID_Range;

  NULL_TYPE_ID : constant Type_ID :=
                      Type_ID(Durra_Interface_Types.NULL_TYPE_ID);

end Durra_Interface;
```

References

[Barbacci 91] Barbacci, M.R.; D.L. Doubleday; C.B. Weinstock; M.J. Gardner; J.M. Wing. *Durra: A Task-Level Description Language Reference Manual (Version 3)* (CMU/SEI-91-TR-18, ADA246405). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1991.

[Doubleday 91] Doubleday, D.L., M.J. Gardner, C.B. Weinstock. *A Description of Cluster Code Generated by the Durra Compiler* (CMU/SEI-91-TR-19, ADA248118). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1991.

Index

application description 10

channel description 7
channel implementation 7
cluster 7
cluster manager 11

link 7

port 7
process 8

task description 7
task implementation 7

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-92-TR-36

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESC-TR-92-036

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (city, state, and zip code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (city, state, and zip code)
ESC/AVS
Hanscom Air Force Base, MA 01731

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESC/AVS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962890C0003

8c. ADDRESS (city, state, and zip code))
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Durra: A Task Description Language User's Manual (Version 2)

12. PERSONAL AUTHOR(S)
Dennis L. Doubleday and Mario R. Barbacci

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month, day)	15. PAGE COUNT
Final	FROM TO	December 1992	80 pp.

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
Durra
task level application description language
distributed Ada applications
developers

19. ABSTRACT (continue on reverse if necessary and identify by block number)

This document describes the use of Durra, a task-level application description language, and its associated toolset. The Durra environment supports the development of highly reconfigurable distributed Ada applications. The intended audience for this document is system managers responsible for Durra installation and Durra application developers.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ SAME AS RPT □ DTIC USERS ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas R. Miller, Lt Col, USAF

22b. TELEPHONE NUMBER (include area code)
(412) 268-7631

22c. OFFICE SYMBOL
ESC/AVS (SEI)

ABSTRACT — continued from page one, block 19